Pro gradu -tutkielma

Tietojenkäsittelytiede

Vaatimusmäärittely ketterässä ohjelmistoprojektissa: menetelmäperusteet ja tapaustutkimus

Juha Louhiranta

1.11.2020

Matemaattis-luonnontieteellinen tiedekunta

Helsingin yliopisto

Ohjaaja(t)

toht. Petri Kettunen

Tarkastaja(t)

toht. Petri Kettunen, prof. Tommi Mikkonen

Yhteystiedot

PL 68 (Pietari Kalmin katu 5)
00014 Helsingin yliopisto

Sähkopostiosoite: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytiede

Tekijä — Författare — Author

Juha Louhiranta

Työn nimi — Arbetets titel — Title

Vaatimusmäärittely ketterässä ohjelmistoprojektissa: menetelmäperusteet ja tapaustutkimus

Ohjaajat — Handledare — Supervisors

toht. Petri Kettunen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Pro gradu -tutkielma	1.11.2020	91 sivua

Tiivistelmä — Referat — Abstract

Tämä pro gradu -tutkielma tarkastelee vaatimusmäärittelyä ketteriä menetelmiä hyödyntävissä ohjelmistoprojekteissa. Tavoitteena on selvittää, miten ohjelmiston vaatimukset muodostuvat ketterien menetelmien avulla ja millaisia vaikutuksia niillä on vaatimusmäärittelyprosessiin. Lisäksi tutkielmassa tarkastellaan, millaisia hyötyjä ketterillä menetelmillä voidaan saavuttaa vaatimusmäärittelyssä ja minkälaisia ongelmia ne saattavat aiheuttaa ohjelmistoprojektin eri vaiheissa. Tutkielmassa myös etsitään eroja ketterien menetelmien ja ketterän vaatimusmäärittelyprosessin sekä perinteisten menetelmien välillä.

Tutkielman tapaustutkimuksen kohteena on Helsingin yliopiston Software Factory -kurssi, jonka aikana toteutettiin asiakasprojekti sulautettua internet-videopuhelujärjestelmää kehittävälle startup-yritykselle. Tapaustutkimuksessa selvitettiin, voidaanko ketterälle vaatimusmäärittelylle tutkimuskirjallisuudessa raportoituja hyötyjä sekä siihen liittyviä haasteita havaita ketterällä menetelmällä toteutettavassa ohjelmistoprojektissa.

Tapaustutkimuksen avulla voitiin empiirisesti havaita kaikki vaatimusmäärittelyn hyödyt sekä yksi epätarkkoihin työmääräarvioihin liittyvä haaste. Tästä voidaan päätellä, että hyötyjen saavuttaminen on mahdollista pienellä tiimillä toteutettavassa asiakasprojektissa, jossa ei ole etukäteen tarkasti määriteltyjä ohjelmiston vaatimuksia. Toisaalta haasteitakin voidaan kohdata jo hyvin lyhyessä ajassa, vaikka rakennettaisiin pientä ja yksinkertaista sovellusta.

ACM Computing Classification System (CCS)

Software and its engineering → Software creation and management → Software development process management → Software development methods → Agile software development

Software and its engineering → Software creation and management → Designing software → Requirements analysis

Avainsanat — Nyckelord — Keywords

vaatimukset, vaatimusspesifikaatio, vaatimusmäärittely, ohjelmistotuotanto, ketterät menetelmät

Säilytyspaikka — Förvaringsställe — Where deposited

Helsingin yliopiston kirjasto

Muita tietoja — övriga uppgifter — Additional information

Ohjelmistojärjestelmien erikoistumislinja

Sisältö

1 Johdanto

Nykyaikaiset, iteratiiviset prosessimallit pyrkivät tuottamaan ohjelmiston noudattaen ketterän ohjelmistokehityksen periaatteita (Agile Alliance, 2001). Ketterien menetelmien tavoitteena on tuottaa asiakkaalle arvoa nopeasti ja reagoida muutostarpeisiin kesken prosessin (B. Boehm ja Turner, 2003). On mahdollista, että ohjelmistoprojektissa tapahtuu muutoksia, jotka voivat koskea vaatimuksia, suunnitelmia, teknologioita tai tiimin kokoonpanoa (Dybå ja Dingsøyr, 2008; Cao ja Ramesh, 2008). Ketterät menetelmät arvostavat erityisesti neljää asiaa: yksilöitä ja heidän keskinäistä kanssakäymistään, toimivaa ohjelmistoa kattavan dokumentaation sijaan, asiakasyhteistyötä sopimusneuvottelujen sijaan sekä kykyä vastata muutokseen (Agile Alliance, 2001). Perinteiset ja suunnitelmavetoiset prosessimallit eroavat ketteristä menetelmistä esimerkiksi tavoissa saavuttaa haluttuja asioita (Dybå ja Dingsøyr, 2008; B. Boehm ja Turner, 2003). Kuitenkin molemmat mallit pitävät sisällään loogisesti samanlaisia kokonaisuuksia, kuten esimerkiksi vaatimusmäärittelyä, kehitystä ja testausta. Molemmissa malleissa on sama haluttu lopputulos: toimiva ohjelma, joka on käyttäjälleen hyödyllinen ja tuottaa hänelle arvoa.

Nykypäivän ohjelmistokehitys on pääosin tiimityöskentelyä. Ohjelmistojen systemaattiseen tuottamiseen tarvitaan teknisten taitojen lisäksi muun muassa ryhmä- ja projektityötaitoja sekä tarkoitukseen sopivaa tuotantoprosessia eli ohjelmistoprosessia (B. Boehm ja Turner, 2003).

Ohjelmistokehityksen yksi oleellisimmista vaiheista on ymmärtää ja määritellä, mitä rakennettavan ohjelmiston tulee tehdä, mitkä ovat sen haluttuja ja olennaisia ominaisuuksia sekä millaisia rajoitteita järjestelmän toimintaan kohdistuu (Sommerville, 2007). Tätä ohjelmistoprosessin vaihetta kutsutaan yleisesti ottaen vaatimusmäärittelyksi. Vaatimusten määrittely ei ole pelkästään tekninen haaste, vaan se on myös kommunikointia asiakkaan, käyttäjien ja ohjelmistokehittäjien välillä. Vaatimuksiin vaikuttavat niin ihmisten mieltymykset, säädökset, standardit kuin käytetty ohjelmistoprosessikin.

Vaatimusmäärittely (*requirements engineering*) on joukko prosesseja, joiden avulla kehitetään ohjelmistojärjestelmän vaatimukset. Prosessit ovat joukko toimintoja, joilla määritellään, dokumentoidaan ja ylläpidetään vaatimuksia. Nämä toiminnot voidaan prosessimielessä jakaa vaatimusten esillesaantiin, analyysin ja neuvotteluun, dokumentointiin, validointiin sekä hallintaan (Kotonya ja Sommerville, 1998).

Ketterä vaatimusmäärittely ymmärretään tämän tutkielman kannalta vaatimusmäärittelyksi, joka tapahtuu osana ketterää ohjelmistoprojektia. Siinä hyödynnetään ketterän ohjelmistokehityksen metodologioiden (kuten XP) ja prosessimallien (kuten Scrum) menetelmiä ja käytänteitä, jotka tukevat ja toteuttavat vaatimusmäärittelyssä perinteisesti olevia prosessin vaiheita. Esimerkiksi Scrum-prosessimallissa sprintin katselmointien kautta toteutetaan vaatimusten validointia (Lucia ja Qusef, 2010). Ketterään vaatimusmäärittelyyn liittyviä käytänteitä ja niiden vaikutuksia on erikseen tunnistettu ja tutkittu (Cao ja Ramesh, 2008; Lucia ja Qusef, 2010; Ramesh et al., 2010; Paetsch et al., 2003; Inayat et al., 2015). Esimerkkejä näistä käytänteistä ovat: kasvokkain tapahtuva kommunikaatio, iteratiivinen vaatimusmäärittely ja katselmointipalaverit (Ramesh et al., 2010). Tämän kaltaisiin käytänteisiin viitataan tässä tutkielmassa ketterän vaatimusmäärittelyn käytänteinä.

Ketterän vaatimusmäärittelyn määritelmä ei ole kuitenkaan yleisesti ottaen täysin selkeä ja yksiselitteinen, sillä ketterät menetelmät eivät itsessään määrittele tarkasti sitä, miten vaatimusmäärittely niissä toteutetaan. Ketterälle vaatimusmäärittelylle on kuitenkin ehdotettu seuraavaa kuvausta (Heikkilä et al., 2015):

> "Ketterässä vaatimusmäärittelyssä vaatimukset saadaan esille, analysoidaan ja spesifioidaan jatkuvassa ja läheisessä yhteistyössä asiakkaan tai asiakasedustajien kanssa, jotta voidaan nopeasti reagoida muuttuviin vaatimuksiin tai ympäristöön. Jatkuva vaatimusten uudelleenarviointi on oleellista rakennettavan järjestelmän menestykselle, ja läheinen yhteistyö asiakkaan tai asiakasedustajien kanssa on oleellinen menetelmä vaatimusten ja järjestelmän validointiin."

Vaatimusmäärittelyä tehdään perinteisissä suunnitelmavetoisissa ohjelmistoprosessimalleissa ja ketterissä menetelmissä toisistaan eroavalla tavalla. Perinteisissä prosesseissa kattava vaatimusmäärittely nähdään usein lineaarisesti etenevän prosessin yhtenä projektin alkupään vaiheena (Sommerville, 2007). Ketterät menetelmät puolestaan iteroivat nopealla syklillä koko ohjelmistokehitysprosessia aina vaatimusten esillesaannista uuden ohjelmistoversion julkaisuun (Heikkilä et al., 2015). Perinteisten ja ketterien prosessien eroista huolimatta myös ketterää vaatimusmäärittelyä tekemällä saadaan suoritettua perinteisen vaatimusmäärittelyprosessiin itsessään liittyvät eri vaiheet ja toiminnot (Paetsch et al., 2003). Ketterän vaatimusmäärittelyn käytänteitä hyödyntämällä on tietyissä tilanteissa mahdollista pienentää vaatimusmäärittelyyn ja erityisesti vaatimuksiin liittyviä ohjelmistoprojektin riskejä (Ramesh et al., 2010). Vastapainona toki on, että tietyissä tilanteissa riskien toteutumisen todennäköisyys voi myös kasvaa. Ketterällä vaatimusmäärittelyllä

voidaan saavuttaa tiettyjä etuja ja hyötyjä perinteiseen vaatimusmäärittelyyn verrattuna (Heikkilä et al., 2015). Hyötyjen vastakohtana on joukko uudenlaisia haasteita ja ongelmia, joilla on toteutuessaan negatiivinen vaikutus kehitystyöhön ja lopputulokseen.

Tässä tutkielmassa tarkastellaan ketteriä menetelmiä vaatimusmäärittelyn ja ohjelmiston vaatimusten näkökulmasta. Tutkielmassa selvitetään ketteriä menetelmiä käsittelevän tutkimuskirjallisuuden pohjalta, miten ohjelmiston vaatimukset syntyvät ohjelmistoprojektissa käytettäessä ketterää vaatimusmäärittelyä ja miten perinteisen vaatimusmäärittelyprosessin eri vaiheet toteutuvat siinä. Tutkielma pyrkii selvittämään, minkälaisia hyötyjä saatetaan saavuttaa tai minkälaisia haasteita tai projektin onnistumiseen vaikuttavia riskejä voidaan kohdata ketterässä vaatimusmäärittelyssä.

Tutkielmaan kuuluvassa tapaustutkimuksessa teoriaa peilataan käytäntöön. Tapaustutkimuksen kohteena on Helsingin yliopiston tietojenkäsittelytieteen laitoksen Software Factory Project -kurssilla toteutettu asiakasprojekti (*Software Factory – University of Helsinki* 2018), jossa kirjoittaja oli myös itse mukana opiskelijana ja kehitystiimin jäsenenä sekä keräämässä aineistoa tätä tutkielmaa varten. Asiakkaana projektissa oli startup-yritys, jolle toteutettiin suorituskykytestausohjelmisto kehitteillä olleelle videopuhelujärjestelmän osalle. Projektissa rakennettiin myös tehokkaampi prototyyppi suorituskykytestauksen kohteena olleesta järjestelmästä. Tässä tutkielmassa selvitetään, miten vaatimusmäärittely toteutui projektissa: miten ketterän vaatimusmäärittelyn esitetyt hyödyt ja haasteet olivat havaittavissa ketterällä ohjelmistokehitysprosessilla tehdyssä projektissa ja miten kohdatut haasteet vaikuttivat projektiin.

1.1 Tutkimusongelma ja -kysymykset

Tämän tutkielman aiheena on vaatimusmäärittely ketterässä ohjelmistoprojektissa. Tutkimusongelma voidaan muotoilla seuraavanlaiseksi kysymykseksi: *Miten vaatimukset muodostuvat ketterässä ohjelmistokehityksessä?*

Tutkittava pääongelma jaetaan tarkemmin seuraaviin tutkimuskysymyksiin:

1. Mitä ketterällä ohjelmistokehityksellä tarkoitetaan?
2. Miten vaatimusmäärittely toteutuu ketterässä ohjelmistokehityksessä?
3. Mitä hyötyjä ja haasteita ketterällä vaatimusmäärittelyllä voidaan havaita perinteisiin menetelmiin verrattuna?

Kaikkia tutkimuskysymyksiä taustoitetaan ensin lähdekirjallisuuden avulla. Sen jälkeen teoriatietoa peilataan tapaustutkimuksen projektiin. Havaintojen avulla selvitetään, saadaanko tapaustutkimuksen tuloksista tukea lähdekirjallisuudessa esitetyille tiedoille ja havaitaanko poikkeuksia tai mahdollisesti jotain uutta.

1.2 Tutkielman rakenne

Tämä tutkielma jakautuu yhteensä kahdeksaan päälukuun. Luvussa 2 esitellään tutkielman aihepiiriin liittyvää aiempaa tutkimuskirjallisuutta, jota on tässä tutkielmassa käytetty pääasiallisena lähteenä. Luvussa 3 käsitellään vaatimusmäärittelyä perinteisen ohjelmistokehityksen näkökulmasta. Ensimmäisenä luvussa esitellään perinteinen ohjelmistokehitysprosessi ja vaatimusmäärittelyn asema siinä. Tämän jälkeen määritellään, mitä vaatimuksella tarkoitetaan, ja lopuksi kuvataan vaatimusmäärittelyn eri vaiheet.

Neljännessä luvussa kuvaillaan ketterää vaatimusmäärittelyä. Ensin käydään yleisesti läpi ketterää ohjelmistokehitystä. Lisäksi ketteristä prosessimalleista XP ja Scrum esitellään tarkemmin. Tämän jälkeen selvennetään, miten perinteisen vaatimusmäärittelyprosessin vaiheet toteutuvat ketterällä vaatimusmäärittelyllä ja minkälaisia hyötyjä ketterällä vaatimusmäärittelyllä voidaan saavuttaa perinteisiin menetelmiin verrattuna. Myös ketterään vaatimusmäärittelyyn liittyvät haasteet käydään läpi. Lopuksi luvussa kerrotaan ketterän vaatimusmäärittelyn käytänteistä sekä niiden vaikutuksista ohjelmistoprojektien vaatimuksiin liittyviin riskeihin.

Viidennessä luvussa esitellään tapaustutkimuksen aineisto ja käytetyt tutkimusmenetelmät. Luvussa kerrotaan myös tarkemmin tutkimusympäristöstä, josta tutkimusaineisto on kerätty.

Kuudennessa luvussa syvennytään tapaustutkimuksen tuloksiin: Millaisia tuloksia saatiin aikaan ja mitkä ovat tärkeimmät havainnot ja käytännön seuraukset? Erityisesti tarkastellaan, oliko tapaustutkimuksen projektissa havaittavissa luvussa 4 esiteltäviä ketterän vaatimusmäärittelyn hyötyjä tai haasteita. Luvussa kerrotaan tarkemmin myös tapaustutkimuksen kohteena olevasta ohjelmistoprojektista, joka toteutettiin asiakkaalle Helsingin yliopiston Software Factory Project -kurssilla.

Seitsemännessä luvussa esitetään tutkielman tuloksiin liittyvä pohdinta. Luvussa myös arvioidaan suoritettua tutkimusta ja esitetään muutama ehdotus siitä, mitä tapaustutkimuksen projektissa itsessään olisi voitu tehdä toisin.

Viimeisessä kahdeksannessa luvussa esitellään tutkielman tulosten perusteella tehtävissä olevat johtopäätökset. Lisäksi esitetään tutkielmassa saadut vastaukset tutkimuskysymyksiin ja arvioidaan tutkimukseen liittyviä heikkouksia, rajoituksia. Lopuksi ehdotetaan muutamia tutkielman kannalta potentiaalisia jatkotutkimusaiheita.

2 Aiempi tutkimuskirjallisuus

Tutkielman aihepiiriin liittyvää aiempaa tutkimuskirjallisuutta on selvitetty kirjallisuuskatsauksen avulla. Kirjallisuuskatsaus on suoritettu noudattaen Jane Websterin ja Richard T. Watsonin artikkelissaan *Analyzing the past to prepare for the future: Writing a literature review* mainitsemaa tekniikkaa (Webster ja Watson, 2002). Aihepiiriin liittyvän aiemmin tehdyn tutkimuksen osalta haluttiin saada selville, miten ohjelmiston vaatimukset ja vaatimusmäärittelyprosessi nähdään sekä perinteisten suunnitelmavetoisten prosessimallien että ketterien menetelmien näkökulmasta. Ketterien menetelmien osalta haluttiin myös selvittää, millaisilla käytänteillä ketterää vaatimusmäärittelyä tehdään sekä minkälaisia hyötyjä ja haasteita ketterään vaatimusmäärittelyyn liittyen on raportoitu.

Kirjallisuuskatsaus tutkielmaa varten suoritettiin tärkeiden aihealueeseen liittyvien lähteiden tunnistamiseksi. Lähdekirjallisuutta on haettu pääasiallisesti digitaalisia hakupalveluja hyödyntäen. Tärkeimpinä hakupalveluina ovat toimineet Google Scholar[†] sekä Helka-hakupalvelu[‡]. Myös muita tieteenalaan keskittyviä palveluita on käytetty potentiaalisina hakupalveluina esimerkiksi IEEE Digital Library ja ACM Digital Library. Hakutermeillä löydetyn tutkimuksen ja aineiston kautta lähdeaineistoa on laajennettu seuraamalla viitteitä eteen- ja taaksepäin. Uutta löytynyttä aineistoa on näin edelleen tutkittu.

Ketteriin menetelmiin ja ketterään vaatimusmäärittelyyn liittyvää aineistoa on haettu erityisesti hakutermeillä: *agile requirements engineering* ja *agile software development*. Ketterän vaatimusmäärittelyn ja siten myös tapaustutkimuksen kannalta seuraavat lähteet on tunnistettu oleellisiksi:

- L. Cao ja B. Ramesh *Agile Requirements Engineering Practices: An Empirical Study* (Cao ja Ramesh, 2008)
- B. Ramesh, L. Cao ja R. Baskerville *Agile requirements engineering practices and challenges: an empirical study* (Ramesh et al., 2010)
- V. T. Heikkilä, D. Damian, C. Lassenius ja M. Paasivaara *A Mapping Study on Requirements Engineering in Agile Software Development* (Heikkilä et al., 2015)

[†]Google Scholar, https://scholar.google.com/
[‡]Helka, http://www.helsinki.fi/helka/

Mainituissa tutkimuksissa esiintyvät tulokset esiintyivät useissa lähteissä viitaten niissä tunnistettuihin ketterän vaatimusmäärittelyn käytänteisiin (Cao ja Ramesh, 2008; Ramesh et al., 2010) ja ketterän vaatimusmäärittelyn hyötyihin ja haasteisiin (Heikkilä et al., 2015).

Myös tietojenkäsittelytieteeseen liittyvää peruskirjallisuutta on käytetty lähteenä tutustuttaessa tarkemmin perinteisiin ohjelmistoprosesseihin, vaatimuksiin ja vaatimusmäärittelyprosessiin. Tunnistettuja lähteitä, joihin myös ketterän vaatimusmäärittelyn tutkimuskirjallisuudessa on viitattu:

- A. van Lamsweerde *Requirements engineering: from system goals to UML models to software specifications* (Lamsweerde, 2009)
- G. Kotonya ja I. Sommerville *Requirements Engineering: Processes and Techniques* (Kotonya ja Sommerville, 1998)
- Kent Beck *Extreme programming explained: embrace change* (Beck, 2000)
- Ian Sommerville *Software Engineering* (Sommerville, 2007)
- B. Boehm ja R. Turner *Balancing agility and discipline: A guide for the perplexed* (B. Boehm ja Turner, 2003)

3 Perinteinen ohjelmistokehitys

Tässä luvussa käydään läpi vaatimusmäärittelyä perinteisen ohjelmistokehityksen näkökulmasta. Perinteistä näkökulmaa edustavat muun muassa Sommerville (2007), Lamsweerde (2009) sekä Kotonya ja Sommerville (1998). Alaluvussa 3.1 kerrotaan perinteisestä ohjelmistokehitysprosessista ja vaatimusmäärittelyn asemasta siinä. Alaluvussa 3.2 määritellään, mitä vaatimuksella tarkoitetaan ohjelmistokehityksessä. Lopuksi alaluvussa 3.3 kuvataan varsinaisen vaatimusmäärittelyprosessin eri vaiheet. Näiden perustietojen pohjalta voidaan seuraavassa luvussa 4 käsitellä ketterää vaatimusmäärittelyä.

Ohjelmiston tuottaminen voidaan abstrahoida joukoksi erilaisia vaiheita ja käytänteitä, joita soveltamalla ohjelmistoprojekti voidaan viedä läpi ja joiden avulla voidaan tuottaa haluttu ohjelmisto systemaattisesti. Näitä abstrahointeja kutsutaan ohjelmistoprosessimalleiksi. Näitä malleja soveltamalla muodostuu ohjelmistoprojektille varsinainen prosessi. Eri prosessimallit lähestyvät ohjelmistokehitystä eri näkökulmista ja ne painottavat eri osa-alueita (Sommerville, 2007; Abrahamsson, Warsta et al., 2003). Kaikkien prosessimallien tarkoituksena on kuitenkin ratkaista mahdolliset ongelmat ja tuottaa toimiva ohjelmisto.

Kaikille ohjelmistoprosesseille yhteisiä vaiheita ja toimintoja ovat (Sommerville, 2007):

- *ohjelmiston määrittely* eli ohjelmiston toiminnallisuuksien ja niitä koskevien rajoitteiden määrittely,
- *ohjelmiston suunnittelu ja implementointi* eli määrittelyn mukaisen ohjelmiston tuottaminen,
- *järjestelmän validointi* eli asiakkaan ohjelmistolta odottaman toiminnallisuuden varmistaminen,
- *järjestelmän evoluutio* eli ohjelmiston mukauttaminen sen muuttuvan toimintaympäristön tarpeisiin.

Ohjelmistoihin ja niiden kehittämiseen liittyvät ohjelmiston vaatimukset. Ohjelmiston vaatimukset kertovat, millaisia ominaisuuksia lopullisessa ohjelmistossa tulisi olla ja miten sen tulisi toimia (Lamsweerde, 2009).

Ohjelmiston vaatimukset voidaan esittää korkeatasoisina käyttäjävaatimuksina sekä tarkempina ja yksityiskohtaisempina järjestelmävaatimuksina (Sommerville, 2007). Vaatimuksia voidaan myös luokitella vaatimusten tyypin mukaisesti erilaisiin ryhmiin, kuten toiminnallisiin ja ei-toiminnallisiin vaatimuksiin (Lamsweerde, 2009; Kotonya ja Sommerville, 1998). Toiminnalliset vaatimukset kertovat, mitä lopullisen ohjelmiston tulee käytännössä sisältää ja miten sen tulee toimia. Ei-toiminnallisia vaatimuksia puolestaan ovat laadulliset vaatimukset ja resurssivaatimukset (Sommerville, 2007; Lamsweerde, 2009; Kotonya ja Sommerville, 1998). Ohjelmiston vaatimusten tuottamista, arviointia ja parantelua voidaan ohjata hyödyntämällä vaatimusten laatukriteereitä, jotka ovat vaatimuksille asetettuja tavoitteita ja hyviksi nähtyjä ominaisuuksia (Lamsweerde, 2009).

Aivan kuten ohjelmistoprosessit voidaan jakaa erilaisiin tiettyjä toimintoja sisältäviin osiin, voidaan varsinainen vaatimusten määrittelykin jakaa (usealla tavalla) erilaisiin vaiheisiin (Hansen et al., 2009). Yksi tapa on jakaa se vaatimusten esillesaantiin, analyysin ja neuvotteluun, dokumentointiin, validointiin sekä hallintaan (Kotonya ja Sommerville, 1998).

3.1 Perinteinen ohjelmistokehitysprosessi

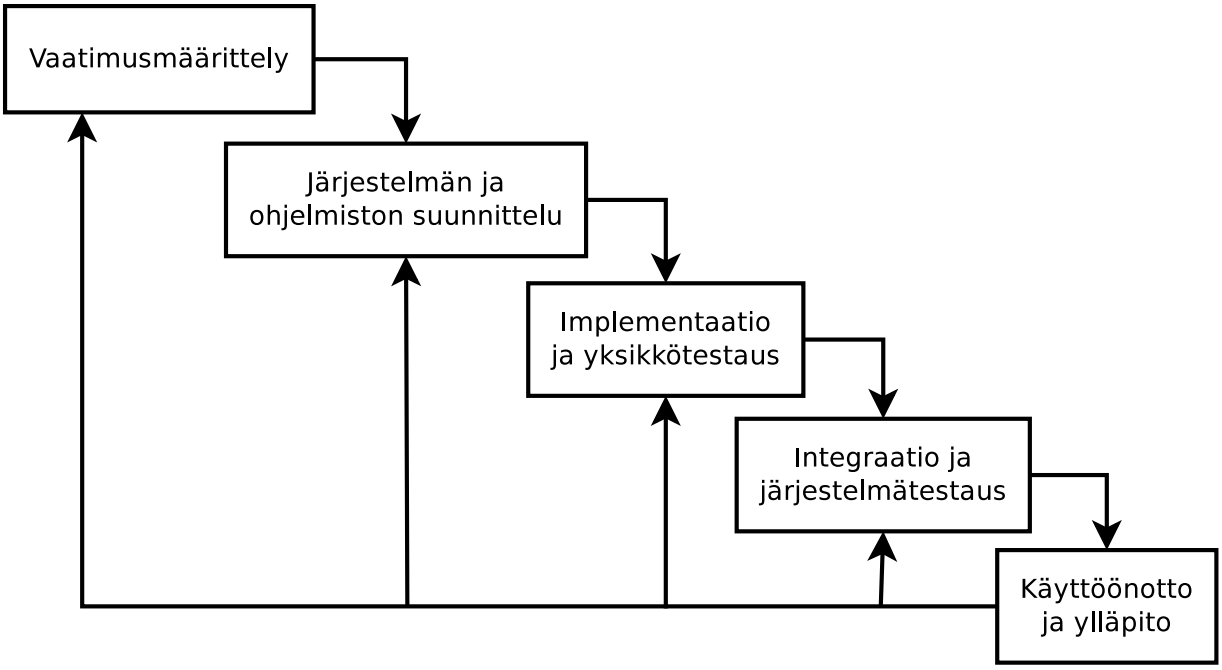
Suunnitelmavetoiset menetelmät käsitetään ohjelmistokehityksen perinteisiksi menetelmiksi (B. Boehm ja Turner, 2003). Perinteiset suunnitelmavetoiset prosessimallit ovat systemaattisia lähestymistapoja, jossa järjestelmän tuottamiseksi noudatetaan tiettyä tarkasti määriteltyä ohjelmistoprosessia (B. Boehm ja Turner, 2003). Jokaisesta prosessin vaiheesta tuotetaan dokumentaatiota, jotta voidaan varmistaa prosessin ja sen vaiheen oikeanlainen suoritus ja jotta voidaan jäljittää tapahtumat tarkasti vaatimusten, suunnittelutyön ja koodin osalta (Royce, 1970; Sommerville, 2007; B. Boehm ja Turner, 2003).

Perinteiset ohjelmistoprosessit määrittelevät ohjelmistokehityksen vaiheet usein hyvin tarkasti. Syynä tähän on se, että vaiheiden huolellisella noudattamisella pyritään takaamaan haluttu lopputulos eli toimiva ja asiakasta tyydyttävä ohjelmisto. Jotta ohjelmistoja voidaan tuottaa uudelleen systemaattisesti samalla tavalla, ohjelmistokehityksen vaiheet ovat tarkkaan määriteltyjä (B. Boehm ja Turner, 2003). Tästä johtuen määritellyn prosessin tarkka noudattaminen itsessään vaatii tarkkuutta.

Perinteisissä menetelmissä ohjelmiston kehityskaari on varsin yksiselitteinen. Ohjelmisto tuotetaan systemaattisesti vaihe kerrallaan. Yhden vaiheen valmistuttua siirrytään seuraavaan, kunnes kaikki vaiheet on käyty läpi ja ohjelmisto on lopulta valmis (Sommerville,

2007). Klassinen esimerkki suunnitelmavetoisesta prosessimallista on vesiputousmalli, jossa edetään vaihe kerrallaan vaatimusmäärittelystä lopulliseen tuotteeseen (Royce, 1970). Muitakin perinteisiä ja suunnitelmavetoisia prosessimalleja on olemassa, kuten esimerkiksi spiraalimalli (Sommerville, 2007; B. W. Boehm, 1988).

Vesiputousmallissa ohjelmistoprosessi on jaettu järjestyksessä suoritettaviin vaiheisiin (Royce, 1970). Nämä vaiheet on esitetty kuvassa 3.1. Seuraavaan vaiheeseen siirtyminen vaatii käytännössä aina sitä, että edellinen vaihe on saatu hyväksytysti päätökseen. Jokaisen vaiheen suoritus dokumentoidaan erikseen. Yksittäisen vaiheen lopputuloksena on yleensä yksi tai useampikin dokumentti, joka aina hyväksytään vaiheen lopuksi ennen siirtymistä seuraavaan vaiheeseen (Royce, 1970; B. Boehm ja Turner, 2003; Sommerville, 2007). Esimerkkidokumentteja ovat ohjelmistovaatimukset tai testaussuunnitelman sisältävät dokumentit.



Kuva 3.1: Vesiputousmallin vaiheet (Sommerville, 2007)

Vesiputousmallin kaksi ensimmäistä vaihetta keskittyvät ohjelmiston suunnitteluun. Malli alkaa *vaatimusten analysointi- ja määrittelyvaiheesta*, jossa päämääränä on koota tiedot tulevan järjestelmän palveluista, rajoitteista ja tavoitteista konsultoimalla järjestelmän tulevia käyttäjiä. Tietojen pohjalta kehittäjät muodostavat yksityiskohtaisen järjestelmän määritelmän. Seuraavassa *järjestelmän ja ohjelmiston suunnitteluvaiheessa* vaatimukset jaetaan joko laitteisto- tai ohjelmistovaatimuksiin. Tämän vaiheen aikana kehittäjät mää-

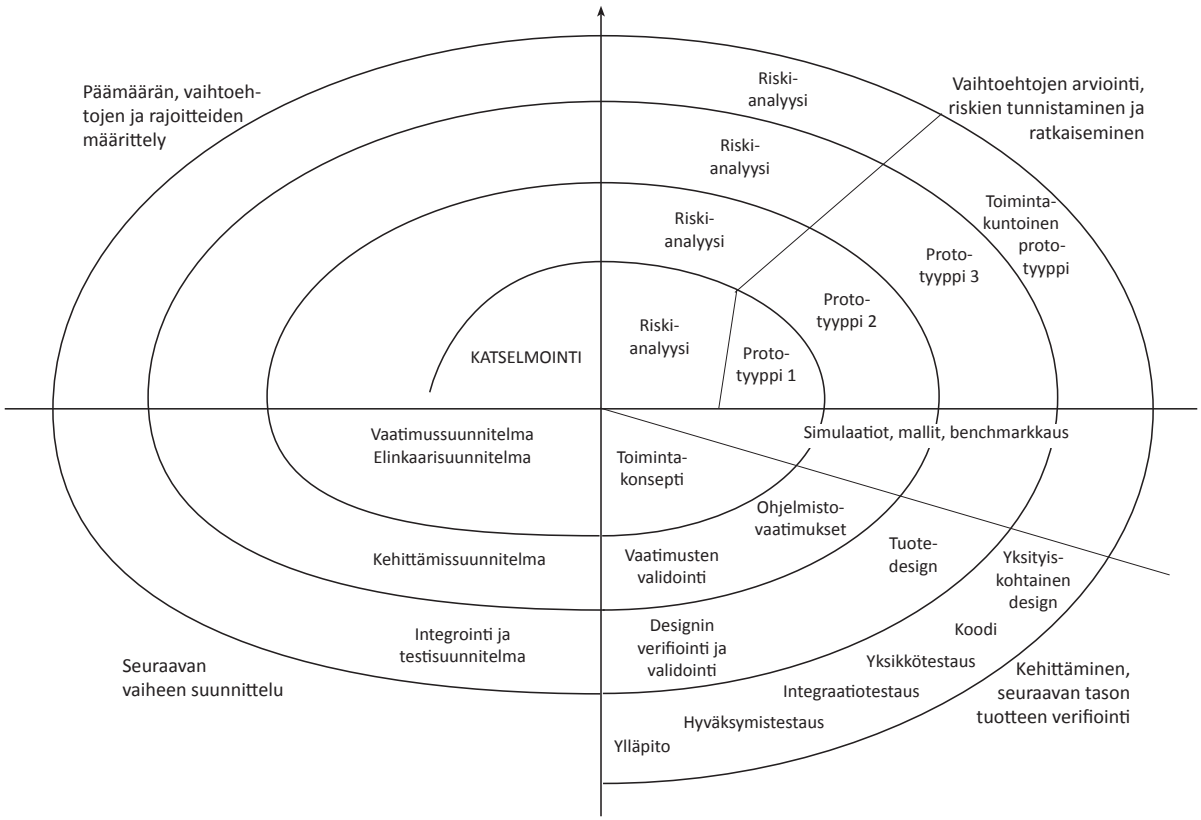
rittelevät yleisen ohjelmistoarkkitehtuurin, keskeiset kokonaisuudet ja niiden väliset suhteet (Sommerville, 2007).

Toteutukseen päästään kolmannessa, *implementaatio- ja yksikkötestausvaiheessa*, jossa suunnitteluvaiheen tuloksien avulla pyritään tuottamaan suunnitelmaa vastaava ohjelmisto. Yksikkötestauksella pyritään varmistamaan ohjelmistolle määriteltyjen spesifikaatioiden täyttyminen. *Integraatio- ja järjestelmätestausvaiheessa* yksittäiset ohjelmiston osat integroidaan keskenään kokonaiseksi järjestelmäksi. Järjestelmätestauksella varmistetaan järjestelmän ohjelmistovaatimuksien täyttyminen. Testauksen jälkeen tuote voidaan toimittaa asiakkaalle (Sommerville, 2007).

Mallin viimeisessä vaiheessa, *käyttöönotto- ja ylläpitovaiheessa* asiakkaalle toimitettu järjestelmä asennetaan ja otetaan todelliseen käyttöön. Ylläpito pitää sisällään muun muassa järjestelmästä kehitysvaiheen jälkeen löytyneiden virheiden korjaamista sekä olemassa olevien ominaisuuksien parantamista uusien vaatimuksien noustessa esiin. Virheiden korjaaminen ja uusien ominaisuuksien tuottaminen voi pitää sisällään aiempien prosessin vaiheiden toistamista (Sommerville, 2007).

Kuten kuvasta 3.1 näkyy, vesiputousmallissa ohjelmiston vaatimukset siis määritellään prosessin ensimmäisessä vaiheessa ennen suunnitteluvaihetta (Royce, 1970; Sommerville, 2007). Vaatimusten tulee olla mahdollisimman kattavia ennen siirtymistä seuraavaan eli suunnitteluvaiheeseen. Jos prosessin aikana huomataan, että johonkin aikaisemman vaiheen tuotokseen tarvitaan muutoksia, tulee prosessi kyseisestä vaiheesta alkaen käydä uudelleen läpi. Jos esimerkiksi kehitysvaiheessa huomataan, että ohjelmiston vaatimuksiin on välttämätöntä tehdä muutoksia, tulee prosessi käydä vaatimusten määrittelyvaiheesta asti uudelleen läpi suorittaen hyväksynnät muillekin vaiheille, kunnes ollaan jälleen kehitysvaiheessa ja voidaan muutokset huomioiden jatkaa siitä, mihin alun perin jäätiin.

Toinen perinteinen, lineaarisesti etenevä prosessimalli on Boehmin esittelemä spiraalimalli (B. W. Boehm, 1988). Spiraalimallissa (kuva 3.2) jokainen kierros vastaa yhtä vaihetta ohjelmiston elinkaaressa. Tällaisia vaiheita ovat esimerkiksi vaatimusten määrittely, järjestelmän kehitys ja järjestelmän testaaminen. Jokainen vaihe eli kierros jakautuu neljään eri vaiheeseen: tavoitteiden määrittelyyn, riskien arviointiin ja pienentämiseen, kehitys- ja validointivaiheeseen sekä arviointi- ja suunnitteluvaiheeseen. Spiraalimallista poikkeavan muihin prosessimalleihin verrattuna tekee sen riskien tunnistamiseen ja niihin varautumiseen keskittyvä vaihe (Sommerville, 2007; B. W. Boehm, 1988).



Kuva 3.2: Spiraalimallin vaiheet kuvattuna (B. W. Boehm, 1988; Sommerville, 2007)

3.2 Vaatimukset ja sen määrittely

Tässä alaluvussa käydään läpi, miten vaatimus on käsitteenä pyritty määrittelemään eri lähteissä. Tämän perusteella muodostetaan käsitys siitä, mitä tarkoitetaan, kun puhutaan ohjelmistovaatimuksista, sekä siitä, miten vaatimus käsitetään tässä tutkielmassa. Lisäksi tarkastellaan sitä, minkälaisiin luokkiin vaatimukset voidaan jakaa sekä minkälaisten laatukriteereiden avulla vaatimuksia voidaan muodostaa ja arvioida.

Vaatimus kuvaa järjestelmän ominaisuutta tai rajoitetta (Kotonya ja Sommerville, 1998). Ne ovat kuvauksia siitä, miten järjestelmän tulee käyttäytyä; tietoa sovelluksen ongelmakentästä; järjestelmän toimintaan liittyviä rajoitteita tai määrittelyjä järjestelmän ominaisuuksista ja piirteistä. Ne voivat olla myös käytettäviin kehitysprosesseihin liittyviä rajoitteita. Kansainvälisen standardin mukaan (*ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary* 2017) vaatimuksella tarkoitetaan ehtoa tai ominaisuutta, jonka käyttäjä tarvitsee ongelman ratkaisemiseksi tai tavoitteen saavutta-

miseksi. Sillä tarkoitetaan myös ehtoa tai ominaisuutta, jonka järjestelmän, järjestelmän osan, tuotteen, palvelun, tuloksen tai komponentin on täytettävä tai omattava sopimuksen, standardin, spesifikaation tai muun muodollisen asiakirjan täyttämiseksi. Vaatimuksella tarkoitetaan standardin mukaan myös edellä mainittujen dokumentoitua esitysmuotoa. Vaatimuksiin kuuluvat sidosryhmien määrälliset ja dokumentoitavat tarpeet, toiveet ja odotukset.

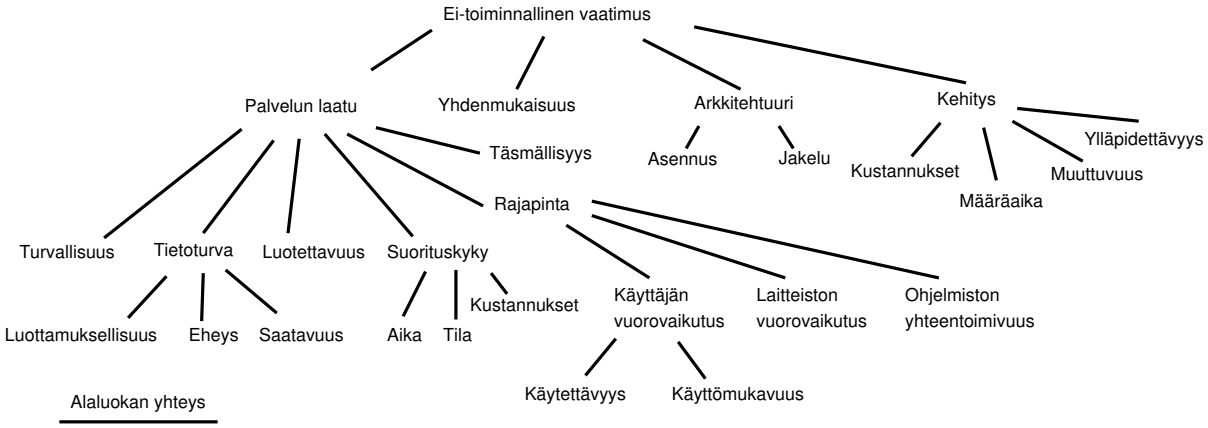
Järjestelmä on kokonaisuus, johon oleellisella tavalla kuuluu joukko tavoitteita, joita sen avulla halutaan saavuttaa. Järjestelmä pystyy saavuttamaan sille asetetut tavoitteet toteuttamalla tavoitteiden tyydyttämiseksi joukon *toiminnallisia palveluita*. Palveluihin liittyy joitakin toimintaympäristöstä tehtyjä *oletuksia*, joiden täytyy päteä palveluiden kunnollisen toiminnan kannalta. Palveluiden täytyy toimia myös tiettyjen *rajoitteiden* (kuten suorituskyky, tietoturva, käytettävyys) mukaisesti. Vastuut tavoitteista, palveluista ja rajoitteista on jaettu järjestelmän eri komponenteille, kuten uudelle toteutettavalle ohjelmistolle, eri rooleissa oleville ihmisille, laitteille ja olemassa oleville ohjelmistoille, joista kolme jälkimmäistä muodostavat uuden ohjelmiston toimintaympäristön (Lamsweerde, 2009).

3.2.1 Toiminnalliset ja ei-toiminnalliset vaatimukset

Vaatimukset luokitellaan usein kahteen kategoriaan: toiminnallisiin ja ei-toiminnallisiin vaatimuksiin (Sommerville, 2007; Kotonya ja Sommerville, 1998). Toiminnalliset vaatimukset kertovat, millaisia palveluita järjestelmän tulee tarjota. Ei-toiminnalliset vaatimukset puolestaan rajaavat sitä, miten nämä palvelut tulee tarjota.

Toiminnalliset vaatimukset määrittelevät ne ominaisuudet, jotka lopullisen ohjelmiston tulee käytännössä sisältää ja joiden mukaan sen tulee toimia. Ne kuvaavat, millaisia palveluita järjestelmä tarjoaa, sekä kertovat, miten tietynlaiseen syötteeseen tulee reagoida ja miten järjestelmän tulee käyttäytyä tietyssä tilanteessa. Toiminnallisilla vaatimuksilla voidaan kuvata myös ei-toivottua toimintaa. Ne riippuvat kehitettävän ohjelmiston tyypistä, oletetuista järjestelmän käyttäjistä ja valitusta lähestymistavasta vaatimusten kirjoittamiseen (Sommerville, 2007). Toiminnallisia vaatimuksia voidaan haluta ryhmitellä karkeammin toiminnallisuuksiin, joita ohjelmiston tulisi tukea. Näitä toiminnallisuuksien yksiköitä kutsutaan joskus myös ominaisuuksiksi (Lamsweerde, 2009). Toiminnalliset käyttäjävaatimukset kuvaavat järjestelmää hyvin abstraktilla tasolla, kun taas toiminnalliset järjestelmävaatimukset ovat hyvin tarkkoja järjestelmän toiminnan yksityiskohdista (Lamsweerde, 2009).

Ei-toiminnalliset vaatimukset määrittelevät ohjelmiston rajoitteet suhteessa siihen, miten toiminnalliset vaatimukset tulee täyttää tai miten ohjelmistoa tulee kehittää (Kotonya ja Sommerville, 1998). Ei-toiminnalliset vaatimukset jakaantuvat useisiin eri alaluokkiin (Sommerville, 2007; Kotonya ja Sommerville, 1998; Lamsweerde, 2009). Lamsweerden mukaan ei-toiminnalliset vaatimukset voidaan jakaa neljään eri pääryhmään, jotka puolestaan voidaan jakaa vielä tarkempiin alaluokkiin. Pääluokat ovat palvelun laatu, yhdenmukaisuus, arkkitehtuurirajoitteet ja kehitysrajoitteet. Kuva 3.3 esittelee Lamsweerden (Lamsweerde, 2009) mainitseman tavan luokitella ei-toiminnallisia vaatimuksia.



Kuva 3.3: Ei-toiminnallisten vaatimuksien luokittelu (Lamsweerde, 2009)

Palvelun laatuun liittyvät vaatimukset määrittelevät laatuun liittyviä ominaisuuksia, joita ohjelmiston toiminnallisilla ominaisuuksilla tulee olla (Lamsweerde, 2009). Laatuvaatimuksia ovat turvallisuus-, tietoturva-, luotettavuus-, suorituskyky-, rajapinta- ja täsmällisyysvaatimukset. Turvallisuusvaatimukset pyrkivät varmistamaan ohjelmiston turvallisen käytön sen toimintaympäristössä, esimerkiksi pyrkimällä estämään onnettomuuksia tai rahallisia menetyksiä. Tietoturvavaatimukset määrittelevät järjestelmään liittyvän omaisuuden ja tiedon suojaamista ympäristön ei-toivotulta toiminnalta. Tietoturvavaatimukset jakautuvat vielä luottamuksellisuus-, yksityisyys-, eheys- ja saatavuusvaatimuksiin. Luotettavuusvaatimukset määrittelevät ohjelmistolle vähimmäisajan, jolloin sen oletetaan toimivan oletetulla tavalla. Tarkkuusvaatimukset rajaavat ja määrittelevät sitä, kuinka tarkasti ohjelmiston käsittelemän informaation tilan on vastattava fyysistä toimintaympäristön tilaa. Suorituskykyvaatimukset määrittelevät ohjelmiston toimintojen kuluttamien resurssien esimerkiksi aikaan ja tilaan liittyviä rajoitteita. Rajapintavaatimukset määrittelevät, miten ohjelmisto toimii siihen liittyvien muiden ympäristön komponenttien kanssa. Rajapintavaatimuksiin kuuluvat käyttäjän vuorovaikutukseen eli käytettävyyteen ja käyt-

tömukavuuteen liittyvät vaatimukset, laitteiston vuorovaikutukseen ja ohjelmiston yhteentoimivuuteen liittyvät vaatimukset.

Yhdenmukaisuusvaatimukset kuvaavat muun muassa lakeihin, kansainvälisiin säädöksiin, sosiaalisiin normeihin, kulttuuriin, politiikkaan, standardeihin liittyviä vaatimuksia ja rajoitteita (Lamsweerde, 2009).

Arkkitehtuurivaatimukset määrittelevät korkealla tasolla ohjelmiston rakenteellisia vaatimuksia tai rajoitteita (Lamsweerde, 2009). Näihin vaatimuksiin kuuluvat myös asennus- ja jakeluvaatimukset, jotka määrittelevät, miten ohjelmiston tulee olla asennettavissa kohdeympäristöihinsä sekä miten ohjelmiston ja siihen mahdollisesti liittyvien päivitysten tulee olla saatavilla. Arkkitehtuurivaatimukset rajaavat mahdollisia ohjelmistoarkkitehtuurivaihtoehtoja.

Kehitysvaatimukset määrittelevät tavat, joilla ohjelmistoa tulee kehittää. Näihin vaatimuksiin kuuluvat kehitystyön kustannukset, toimitusaikataulut, toimintojen muuttuvuus, ohjelmakoodin ylläpidettävyys, uudelleenkäytettävyys ja siirrettävyys. Kehitysvaatimukset eivät varsinaisesti rajoita tai määrittele sitä, miten ohjelmiston toiminnalliset vaatimukset tulee täyttää (Lamsweerde, 2009). Kehitysvaatimuksiin voi sisältyä myös vaatimuksia esimerkiksi tietyn kehitysprosessin noudattamisesta, suunnittelutyökalun käyttämisestä sekä tietyn laatustandardin täyttämisestä prosessissa (Sommerville, 2007).

Vaatimusten luokittelussa voi tietyissä tilanteissa ilmetä päällekkäisyyttä, ja sama vaatimus voidaankin nähdä kuuluvan niin toiminnallisiin kuin ei-toiminnallisiinkin vaatimuksiin (Kotonya ja Sommerville, 1998). Sama vaatimus voi myös kuulua tietyissä tilanteissa useaan eri ei-toiminnallisten vaatimusten alaluokkaan (Lamsweerde, 2009). Vaatimusten avulla voidaan ilmaista haluttuja, ei-hyväksyttyjä ja mieluisimpia toimintoja (Sommerville, 2007). Toiminnallinen vaatimus määrittelee usein vain yhtä toimintoa. Yksittäinen ei-toiminnallinen vaatimus saattaa kuitenkin liittyä usean eri toimintoon ja määrittää niiden toteuttamiselle rajoitteita (Kotonya ja Sommerville, 1998). Vaatimusten luokittelun avulla voidaan myös tarvittaessa pyrkiä tunnistamaan mahdollisesti huomaamatta jääneitä tai puuttuvia vaatimuksia.

3.2.2 Vaatimusten laatukriteerit

Vaatimusdokumentaatiossa esitettyjä vaatimuksia ja niiden soveltuvuutta voidaan arvioida niille asetettavien laatutekijöiden kautta (Lamsweerde, 2009). Laatutekijät ovat vaatimusmäärittelyn vaatimuksille asetettuja tavoitteita. Tuotettavia vaatimuksia voidaan

arvioida ja parannella näiden kriteerien avulla. Laatukriteereiden täyttymisen varmistaminen vie aikaa (B. Boehm, 2000). Joidenkin kriteerien varmistaminen voi olla hankalaa, koska niiden tavoitteet itsessään voivat olla epäsuoria, epäselviä tai määrittelemättömiäkin. Tällaisia kriittisiä ominaisuuksia ovat esimerkiksi kattavuus, tarkoituksenmukaisuus ja asiaankuuluvuus (Lamsweerde, 2009).

Laatukriteereiden vaillinaisella täyttymisellä voi olla kohtalokkaita seurauksia tuotettavan järjestelmän laatuun. Parhaassa tapauksessa laatukriteereiden täyttymättömyys johtaa ainoastaan turhaan työhön ja kasvaneeseen vaatimusmäärittelylle luontaiseen riskiin (Lamsweerde, 2009). Vaatimuksille mainittuja hyviä laatukriteereitä on listattu taulukossa 3.1.

Laatukriteeri	**Kuvaus**
Kattavuus	Rakennettavalla järjestelmällä pitää voida toteuttaa kaikki sen tavoitteet. Tavoitteiden itsessään täytyy olla selvillä. Uuden järjestelmän täytyy kattaa kaikki sen vaatimukset ilman ei-toivottuja ominaisuuksia.
Johdonmukaisuus	Vaatimusten täytyy olla saavutettavissa keskenään eli niiden täytyy olla yhteensopivia.
Tarkoituksenmukaisuus	Vaatimusten täytyy kuvata uuden järjestelmän todelliset tarpeet. Ohjelmistovaatimusten täytyy kuvata riittävän hyvin järjestelmävaatimuksia. Ongelmakentän lainmukaisuudet täytyy esittää oikein sekä ympäristöoletusten täytyy olla realistisia.
Yksiselitteisyys	Vaatimukset täytyy muotoilla tavalla, joka estää monitulkintaisuuden. Jokainen termi täytyy määritellä ja niitä tulee käyttää johdonmukaisesti.
Mitattavuus	Vaatimus täytyy muotoilla tarkkuudella, jolla sitä voidaan verrata muihin vaihtoehtoihin, jolla sitä kyetään testaamaan sekä jolla voidaan varmistaa sen täyttyminen implementaatiosta.
Asiaankuuluvuus	Kaikkien vaatimusten täytyy tukea yhden tai useamman järjestelmän päämäärän saavuttamista. Niiden täytyy kuvata ongelmakentän elementtejä pelkän toteutuksen sijaan.
Soveltuvuus	Vaatimuksen täytyy olla realistinen budjetin, aikataulun ja teknologiarajoitteiden valossa.
Ymmärrettävyys	Vaatimusten täytyy olla ymmärrettäviä niitä käyttäville henkilöille.
Hyvä rakenne	Vaatimusdokumentaation täytyy olla järjestetty niin, että se korostaa yhteyksiä eri osien välillä.
Muokattavuus	Vaatimuksia täytyy olla mahdollista muokata, lisätä tai poistaa siten, että vaikutus on mahdollisimman paikallinen.
Jäljitettävyys	Asiayhteys, johon vaatimus liittyy, tai sen muokkaamisen syy tulee olla selvitettävissä. Asiayhteyden tulee sisältää perustelut luonnille, muokkaukselle tai käytölle. Vaatimuksen poistamisen vaikutusten tulee olla helposti arvioitavissa esimerkiksi liittyvien vaatimusten, olemassa olevan testidatan ja ohjelmakoodin osalta.

Taulukko 3.1: Vaatimusten hyviä laatukriteereitä (Lamsweerde, 2009)

Laatutekijöillä on myös vastakohtinaan ei-toivottuja ominaisuuksia, joita tulisi välttää vaatimusmäärittelyä tehtäessä. Vaatimusten heikkoudet ja virheet koostuvat hyvien puolien vastakohdista. Vaatimuksien heikosta laadusta kertovia ominaisuuksia on listattu taulukossa 3.2.

Vaatimuksen heikkous tai virhe	**Kuvaus**
Vaatimusten sivuuttaminen	Järjestelmälle oleellinen tavoite, vaatimus tai oletus jää mainitsematta.
Ristiriita	Vaatimus on ristiriidassa toisen vaatimuksen kanssa, minkä seurauksena niitä ei voida täyttää samanaikaisesti.
Sopimattomuus	Vaatimus ei tarkoituksenmukaisesti määrittele järjestelmän todellista tarvetta.
Monitulkintaisuus	Vaatimus voidaan tulkita monella eri tavalla.
Heikko mitattavuus	Vaatimusta ei voida verrata vaihtoehtoisiin toteutuksiin. Vaatimuksen testaaminen tai varmistaminen toteutetusta järjestelmästä ei ole mahdollista.
Kohina	Vaatimus ei sisällä tietoa mistään ongelmakentän ominaisuudesta.
Ylimäärittely	Vaatimuksen määrittely kuvaa teknistä toteutusta eikä pidättäydy pelkässä ongelmakentän kuvauksessa.
Toteuttamiskelvottomuus	Vaatimusta ei voida toteuttaa budjetin, aikataulun ja käytettävien teknologioiden valossa.
Vaikeaselkoisuus	Vaatimus on määritelty vaikeaselkoisesti sen henkilön kannalta, jonka tulisi se ymmärtää.
Heikko rakenne	Vaatimuksia ei ole järjestelty minkään järjellisen tai näkyvän järjestelysäännön perusteella.
Eteenpäin viittaus	Vaatimuksessa hyödynnetään sellaisia ongelmakentän ominaisuuksia, joita ei ole vielä määritelty.
Myöhään määrittely	Vaatimus kuvaa ongelmakentän ominaisuutta liian myöhään tai sattumalta.
Heikko muokattavuus	Vaatimuksen muokkaaminen saattaa aiheuttaa muutoksia useaan muuhunkin vaatimukseen.
Jäljittämättömyys	Vaatimuksen perussyyt, muokkaukset ja riippuvuudet eivät ole selvitettävissä.

Taulukko 3.2: Vaatimuksien laatuun liittyvät heikkoudet ja virheet (Lamsweerde, 2009)

3.3 Vaatimusmäärittelyprosessi

Tässä luvussa esitellään vaatimusmäärittelyn eri vaiheet niin kuin ne yleensä nähdään perinteisessä tutkimuskirjallisuudessa. Ohjelmiston vaatimusmäärittelyprosessi voidaan jakaa toisistaan eroaviin ja eri toimintoja sisältäviin vaiheisiin. Lähteestä riippuen vaatimusmäärittelyprosessi on voitu jakaa esimerkiksi kahdesta aina seitsemään eri osaan

(Hansen et al., 2009). Tämän tutkielman kannalta vaatimusmäärittelyprosessi käsitetään viidestä erityyppisestä vaiheesta koostuvana prosessina. Nämä vaiheet ovat vaatimusten esillesaanti, analyysi ja neuvottelu, dokumentointi, validointi sekä hallinta (Kotonya ja Sommerville, 1998). Tämä lähestymistapa on valittu, sillä se on varsinkin ketterään vaatimusmäärittelyyn liittyvässä tutkimuksessa usein käytetty jakotapa (Paetsch et al., 2003; Ramesh et al., 2010; Heikkilä et al., 2015; Lucia ja Qusef, 2010). Yhtenäinen tapa käsittää vaatimusmäärittelyprosessi helpottaa perinteisen ja ketterän lähestymistavan vertaamista toisiinsa.

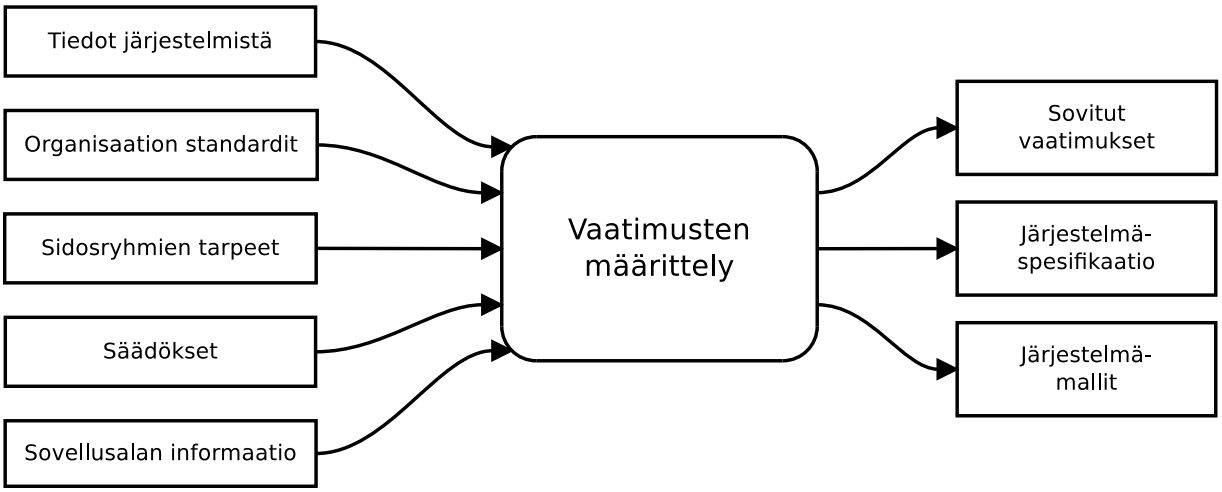
Vaatimusmäärittely on mahdollista tiivistää kolmen eri vaatimuksiin liittyvän ulottuvuuden selvittämiseen: *miksi* uusi järjestelmä tarvitaan, *mitä* tarpeita sen avulla täytetään ja *kuka* näiden tarpeiden täyttämiseen osallistuu uudessa järjestelmässä (Lamsweerde, 2009). *Miksi*-ulottuvuuden kautta pyritään selvittämään, mitkä ovat niitä ongelmia, joita kehitettävällä ohjelmistolla halutaan ratkaista. *Mitä*-ulottuvuuden avulla pyritään ymmärtämään, millaisia toiminnallisia palveluita kehitettävän järjestelmän täytyy toteuttaa. *Kuka*-ulottuvuuden kautta pyritään selvittämään järjestelmän eri osien vastuualueet, jotta järjestelmän tavoitteet, palvelut ja rajoitteet voidaan saavuttaa.

Vaatimusten määrittely ei ole täysin suoraviivaista ja helppoa (Lamsweerde, 2009). Perinteisten tai ketterien menetelmien käyttö aiheuttavat omia erojaan vaatimusmäärittelyprosessin kulkuun (Cao ja Ramesh, 2008). Prosessimallista riippuen vaatimukset voidaan esimerkiksi luoda iteratiivisesti ja inkrementaalisesti prosessin aikana tai määritellä tarkasti heti projektin alussa ennen varsinaisen kehitystyön aloittamista.

Vaatimusten tuottaminen on monimutkainen prosessi, minkä takia vaatimusmäärittely on aina riskialtis osa järjestelmän suunnittelutyötä. Täten myös ongelmat ovat mahdollisia. Ohjelmiston parissa työskentelevien ihmisten tiedon prosessointikyvyillä on rajoitteensa ja kehittäjien ja projektin sidosryhmien välinen vuorovaikutus on usein monimutkaista (Sommerville ja Sawyer, 1997). Onkin tärkeää, että kehittäjien ja sidosryhmien välit ovat hyvät, sillä puutteellinen kommunikaatio on este tehokkaalle vaatimusten määrittelylle (Ramesh et al., 2010; Hansen et al., 2009). Asiakas ei aina itsekään tiedä tai ymmärrä, mitä hän oikeastaan haluaa, mitä ohjelmiston kuuluisi varsinaisesti tehdä tai mitä sen avulla tulisi saavuttaa (Orr, 2004). Vaatimusmäärittely myös reagoi herkästi, mikäli organisaatiossa joudutaan tekemään muutoksia ohjelmiston kehittämiseen liittyvään resursointiin ja päätöksentekoon (Hansen et al., 2009).

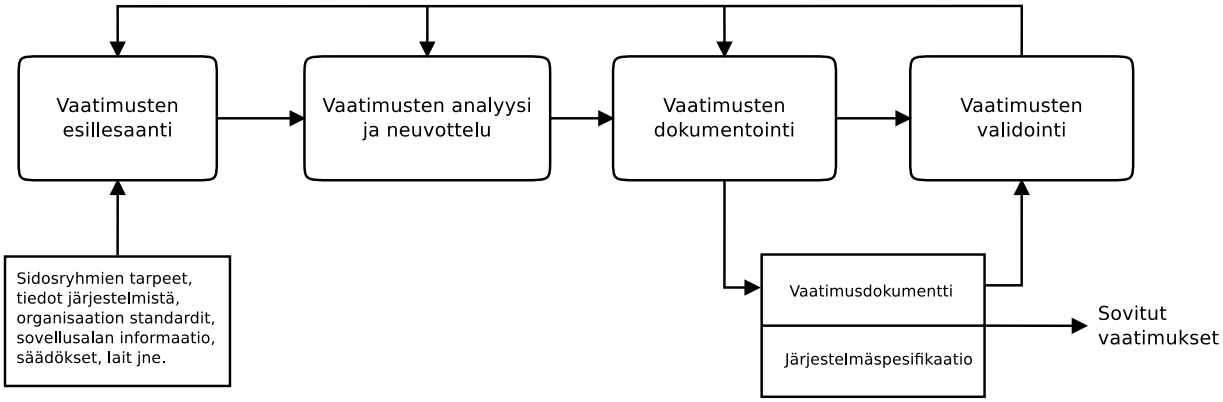
Vaatimusmäärittelyprosessi on usein hieman erilainen eri organisaatioiden välillä, mutta sen tietolähteiden tyypit ja tulokset ovat samankaltaisia. Ohjelmistoprojektissa eri sidos-

ryhmien tarpeet nähdään usein tärkeimpinä lähteenä vaatimuksille. Nämä tarpeet ovat kuitenkin vain osa siitä kaikesta informaatiosta, jota vaatimusmäärittelyprosessiin yleensä tarvitaan ja saadaan. Kuvassa 3.4 on esitetty yksi tapa kuvata vaatimusmäärittelyn erityyppisiä tietolähteitä sekä vaatimusmäärittelyn tuloksia. Erilaisia tietolähteitä on kuvassa esitelty viisi. Vaatimusmäärittelyn tueksi voidaan kerätä tietoa korvattavasta järjestelmästä tai toteutettavaan järjestelmään liittyvistä muista järjestelmistä. Myös sovellusalaan ja järjestelmän ympäristöön liittyvää yleistä informaatiota tarvitaan. Erilaiset lait ja säädökset esimerkiksi tietoturvaan liittyen on myös otettava huomioon. Prosessiin ja vaatimuksiin vaikuttavat myös asiakas- ja toimittajaorganisaation standardit ja käytännöt. Ne asettavat vaatimuksia esimerkiksi kehitysmenetelmiin tai laadunvalvontaan. Vaatimusmäärittelyn tuloksena saadaan eri sidosryhmien hyväksymät vaatimukset. Joissakin tapauksissa vaatimusten määrittely tuottaa myös tarkempia järjestelmäspesifikaatioita tai -malleja (Kotonya ja Sommerville, 1998).



Kuva 3.4: Vaatimusmäärittelyn erityyppiset tietolähteet ja tulokset (Kotonya ja Sommerville, 1998)

Kotonya ja Sommerville (1998) kuvasivat mallin yleiselle systemaattiselle vaatimusmäärittelyprosessille ja vaatimusmäärittelydokumentaation tuottamiselle. Prosessi on esitetty kuvassa 3.5. Tämä prosessi koostuu neljästä vaiheesta, jotka ovat vaatimusten esillesaanti, vaatimusten analyysi ja neuvottelu, vaatimusten dokumentointi ja vaatimusten validointi. Lisäksi tämän vaatimusmäärittelyprosessin rinnalla toimii vaatimusten hallinta, jolla muutoksia vaatimuksiin voidaan hallita. Nämä kaikki vaiheet käsitellään seuraavaksi tarkemmin omissa alaluvuissaan.



Kuva 3.5: Vaatimusmäärittelyprosessin eri vaiheet (Kotonya ja Sommerville, 1998)

3.3.1 Esillesaanti

Vaatimusten esillesaanti on vaatimusmäärittelyprosessin ensimmäinen vaihe. Esillesaantivaiheen aikana kehittäjät pyrkivät luomaan itselleen käsityksen ohjelmiston tulevasta toimintaympäristöstä. Se pitää sisällään kaikki vaatimusten löytämiseen liittyvät toiminnot. Järjestelmän vaatimukset selvitetään sidosryhmien avustuksella sekä tarkastamalla muita tietolähteitä. Vaihe pitää sisällään kaikkien oleellisten osapuolten ja tietolähteiden tunnistamisen. Vaatimusten esillasaannissa on oleellista ymmärtää tulevan sovelluksen toimintakentän ominaisuudet, mahdolliset liiketoimintatavoitteet, järjestelmän rajoitteet, sidosryhmät sekä heidän tarpeensa, jotta saadaan ymmärrys kehitettävästä järjestelmästä (Paetsch et al., 2003; Hansen et al., 2009; Kotonya ja Sommerville, 1998).

Vaatimusten esillesaantia tukevia tekniikoita ovat esimerkiksi haastattelut, käyttötapauskuvaukset, havainnointi, fokusryhmähaastattelu ja aivoriihet (Paetsch et al., 2003). Haastatteluiden avulla voidaan selvittää kehitettävää järjestelmää koskevaa tietoa ja mielipiteitä potentiaalisilta käyttäjiltä ja sidosryhmiltä. Haastatteluilla voidaan myös korjata virheitä ja väärinymmärryksiä. Käyttötapauskuvausten avulla voidaan kuvata, mitä käyttäjät kykenevät järjestelmällä tekemään. Havainnoinnin avulla tutkitaan käyttäjien työskentelyä suoraan tai epäsuorasti tehden tapahtumasta muistiinpanoja. Fokusryhmähaastatteluissa voidaan vapaamuotoisen pienryhmäkeskustelun avulla selvittää käyttäjien tarpeita sekä käsityksiä tärkeistä ja halutuista ominaisuuksista. Myös käyttäjien kohtaamia mahdollisia ongelmia ja haasteita voidaan selvittää. Aivoriihien avulla voidaan kehittää luovia ratkaisuja kohdattuihin haasteisiin. Prototyyppi on järjestelmä alustava tai kokeellinen versio, jonka avulla voidaan saada esiin uusia vaatimuksia tai validoida tiedossa olevia.

3.3.2 Analyysi ja neuvottelu

Analyysivaiheessa aiemmin kerätyt vaatimukset käydään tarkasti läpi. Lisäksi pyritään tunnistamaan kaikki niihin liittyvät ongelmat sekä ratkaisut näihin ongelmiin. Eri sidosryhmien kanssa käydyissä neuvotteluissa pyritään pääsemään yhteiseen sopimukseen ja kompromissiin siitä, mitkä näistä vaatimuksista hyväksytään (Kotonya ja Sommerville, 1998).

Analyysin aikana vaatimuksista tarkistetaan niiden tarpeellisuus ja keskinäinen ristiriidattomuus. Vaatimusten pitää kattavasti kuvata halutut palvelut ja rajoitteet eli niissä ei saa esiintyä puutteita. Vaatimusten täytyy myös olla soveltuvia esimerkiksi suhteessa budjettiin. Useista lähteistä kerätyissä vaatimuksissa ilmenee usein ristiriitoja. Vaatimuksissa voi myös ilmetä puutteita tiedon tai kokonaan puuttuvien vaatimusten muodossa. Määritellyt vaatimukset voivat myös olla toteuttamiskelvottomia projektin käytettävissä oleviin resursseihin verrattuna. Analyysivaihetta suoritetaan usein myös samanaikaisesti esillesaantivaiheen kanssa, sillä usein löydettäessä uusia vaatimuksia niiden analysointi tapahtuu osittain välittömästi (Kotonya ja Sommerville, 1998).

Vaatimusten analysointia tukevia tekniikoita ovat muun muassa vaatimusten priorisointi, mallinnus ja JAD-tilaisuudet (Paetsch et al., 2003). Vaatimusten priorisoinnin avulla eniten arvoa tuottavat ominaisuudet voidaan asiakkaan toimesta priorisoida etusijalle, jolloin arvoa saadaan tuotettua varhain ja jolloin pienemmällä prioriteetilla olevat vaatimukset voidaan jättää toteuttamatta resurssien loppuessa. Mallinnuksen avulla voidaan kuvata järjestelmää useilla eri tavoilla, esimerkiksi mallintamalla tiedon liikkumista järjestelmässä tai järjestelmien välillä. Joint Application Development (JAD) on strukturoitu tilaisuus, jossa asiakkaat ja kehittäjät yhdessä keskustelevat tuotteen toivotuista ominaisuuksista.

3.3.3 Dokumentointi

Vaatimusmäärittelydokumentaation tarkoitus on kommunikoida vaatimukset eri sidosryhmien ja kehittäjien välillä. Vaatimukset dokumentoidaan sopivalla tarkkuudella ja niiden täytyy olla ymmärrettävissä kaikkien järjestelmän sidosryhmien välillä. Vaatimusmäärittelydokumentaatio toimii perustana vaatimusmäärittelyä seuraavien tulosten ja vaiheiden arvioinnille sekä vaatimusten hallinnalle. Vaatimusmäärittelydokumentaatio voidaan liittää myös osaksi asiakkaan ja toimittajan välistä sopimusta (Paetsch et al., 2003; Kotonya ja Sommerville, 1998). Hyvän vaatimusdokumentaation vaatimukset noudattavat vaatimusten laatukriteereitä (katso luku 3.2.2).

3.3.4 Validointi

Vaatimusten validointi on vaatimusmäärittelyprosessin viimeinen vaihe. Validoinnin tarkoituksena on varmistaa, että määritellyt vaatimukset ovat hyväksyttävä kuvaus tuotettavasta järjestelmästä. Eri sidosryhmät ja kehittäjät tutkivat vaatimukset ongelmien, puutteiden ja epäselvyyksien varalta sekä varmistavat, että vaatimukset ovat johdonmukaisia ja kattavia eli kaikki järjestelmän vaatimukset kattavia. Validoinnissa on tarkoitus huomata ongelmat ennen vaatimusmäärittelydokumentin käyttämistä toteutuksen pohjana. Vaatimusmäärittelydokumentin ja siinä listattujen vaatimusten tulisi noudattaa määriteltyjä laatustandardeja ja validointiprosessin tulisi keskittyä validoimaan sitä, miten vaatimukset on dokumentissa kuvattu. Tuloksena validoinnista on yleensä joko lista huomatuista ongelmista ja toimenpiteistä, joita tunnistetut ongelmat edellyttävät, tai hyväksyntä kaikilta projektin sidosryhmiltä (Paetsch et al., 2003; Kotonya ja Sommerville, 1998).

Analyysin ja validoinnin ero voidaan käsittää siten, että analyysivaiheessa käydään läpi sidosryhmiltä ja muista lähteistä esille saatuja raakoja vaatimuksia. Niissä voi olla vielä paljon puutteita ja epämuodollisia ilmaisuja. Validoinnissa tarkistetaan valmis vaatimusmäärittelydokumentti, jonka pitää sisältää kaikki järjestelmän vaatimukset ilman minkäänlaisia ristiriitoja tai puutteita. Analyysissä selvitetään, onko määritelty oikeita vaatimuksia, kun taas validoinnissa tarkastellaan, ovatko vaatimukset oikein määriteltyjä (Kotonya ja Sommerville, 1998).

3.3.5 Hallinta

Vaatimusten hallinnan avulla seurataan muutoksia järjestelmän vaatimuksiin ja varmistetaan, että muutokset tehdään kontrolloidusti. Vaatimusten hallinnalla uusiin, poistuviin tai muuttuviin vaatimuksiin voidaan reagoida oikealla tavalla ja näin toimittaa asiakkaalle hänen tarpeitaan vastaava ohjelmisto. Vaatimusten hallinta tapahtuu rinnakkain vaatimusmäärittelyprosessin muihin vaiheisiin nähden ja sitä jatketaan muun kehityksen aikana, sillä vaatimukset voivat muuttua missä tahansa kehitysprosessin vaiheessa ja näitäkin muutoksia täytyy hallita. Muutoksia vaatimuksiin voivat aiheuttaa esimerkiksi havaitut virheet ja puutteet vaatimuksissa, ulkoiset tekijät, uudet säädökset ja lait, sidosryhmien ymmärryksen lisääntyminen ja muuttuvat liiketoiminnan prioriteetit. Vaatimusten hallintaan liittyy myös vaatimusten jäljitettävyys. Jäljitettävän vaatimuksen suhteet muihin vaatimuksiin, suunnitelmiin, toteutukseen ja dokumentaatioihin voidaan selvittää ja muutoksen vaikutukset voidaan arvioida (Paetsch et al., 2003; Kotonya ja Sommerville, 1998).

4 Ketterät menetelmät ja vaatimusmäärittely

Tässä luvussa käydään läpi ketterässä ohjelmistokehityksessä tapahtuvaa vaatimusmäärittelyä. Ensimmäiseksi luvussa 4.1 tarkastellaan, mitä ketterällä ohjelmistokehityksellä tarkoitetaan. Luvussa 4.2 esitellään ketteristä prosessimalleista XP ja Scrum huomioiden myös, miten vaatimusmäärittely niissä toteutuu. Lopuksi luvussa 4.3 keskitytään varsinaisesti ketterään vaatimusmäärittelyyn. Ensin käydään läpi tarkemmin vaatimusmäärittelyprosessin toteutuminen ketterissä projekteissa. Tämän jälkeen tarkastellaan ketterällä vaatimusmäärittelyllä mahdollisesti saavutettavia hyötyjä sekä siinä kohdattavia haasteita. Tämän jälkeen listataan tutkimuskirjallisuudessa tunnistettuja ketterän vaatimusmäärittelyn käytänteitä sekä selvennetään, miten näiden käytänteiden hyödyntäminen vaikuttaa ohjelmistoprojektin vaatimuksiin liittyviin riskeihin.

4.1 Ketterä ohjelmistokehitys

Perinteisten tarkasti määriteltyjen ohjelmistoprosessien, joita esiteltiin luvussa 3, rinnalle on kehittynyt joukko ketteriä menetelmiä selviämään haasteista, joihin perinteiset menetelmät eivät ole kyenneet vastaamaan (Abrahamsson, Warsta et al., 2003). Perinteisiä menetelmiä on kuvattu ketterien näkökulmasta liian raskaina ja kankeina. Ketterien menetelmien päätavoitteena on tuottaa toimiva järjestelmä mahdollisimman aikaisessa vaiheessa kehitystyötä (Ramesh et al., 2010). Internetin mahdollistamat nopeat ja useat julkaisut ovat myös tehneet pitkäkestoisista ja jäykistä prosesseista tietyissä tilanteissa kannattamattomia (Heikkilä et al., 2015).

Ketterälle ohjelmistokehitykselle ei ole täysin yksiselitteistä ja tarkkaa määritelmää. Sen ominaisuuksia on kuvattu monella eri tavalla, mutta niissä toistuvat kuitenkin samankaltaiset piirteet. Abrahamsson, Salo et al. (2002) määrittelivät moneen aiempaan määritelmään pohjautuen tutkimuksessaan ohjelmistokehitysmenetelmän ketteräksi, jos se on inkrementaalinen, yhteistyöhön kannustava, suoraviivainen ja mukautuva. Inkrementaalisessa prosessissa ohjelmiston julkaisut ovat kooltaan pieniä sekä prosessin iteraatiot ovat lyhyitä, jolloin julkaisuja voidaan tehdä usein. Yhteistyöhön kannustavassa prosessissa

asiakas ja kehittäjät työskentelevät jatkuvasti yhdessä tiiviisti kommunikoiden. Suoran kommunikaation avulla selvitetään ja määritellään projektin tavoitteet sekä ratkaistaan projektin aikana ilmenevät ongelmat. Suoraviivainen prosessi on hyvin dokumentoitu, helposti kaikkien sitä hyödyntävien opittavissa ja itse prosessi on muokattavissa vastaamaan erilaisia tarpeita. Mukautuva prosessi pystyy reagoimaan nopeasti muutostarpeisiin, jotka kohdistuvat kehitettävään ohjelmistoon sekä prosessiin itseensä. Mukautuvalla prosessilla voidaan toteuttaa muutoksia kesken projektin, jolloin tuotettavan ohjelmiston kehitystä voidaan ohjata oikeaan suuntaan sekä prosessia itseään voidaan mukauttaa, jolloin se saadaan tuottamaan parempia tuloksia ja toimimaan tehokkaammin.

Näitä ketteryyden kriteereitä täyttäviä prosessimalleja ovat muun muassa Extreme Programming (XP), Scrum, Kanban, Crystal Methods, Feature Driven Development, Rational Unified Process, Dynamic Systems Development Method ja Adaptive Software Development (Abrahamsson, Salo et al., 2002).

Ketterät menetelmät jakavat ohjelmiston kehityskaaren lyhyisiin, yleensä yhdestä neljään viikkoa kestäviin iteraatioihin (Abrahamsson, Salo et al., 2002). Jokaisen iteraation aikana pyritään toimittamaan aina suurimman prioriteetin sillä hetkellä omaavat ominaisuudet. Jokainen iteraatio on kuin pieni projekti itsessään. Sen aluksi suunnitellaan ja sovitaan asiakkaan kanssa tulevan syklin aikana toteutettavat toiminnallisuudet. Iteraation päätteeksi uusi versio ohjelmistosta esitellään ja toimitetaan asiakkaalle.

Suora kommunikaatio on tärkeä työkalu kehittäjien, asiakkaan ja muiden projektiin liittyvien sidosryhmien välillä (Ramesh et al., 2010). Hyvä kommunikaatio on myös oleellista ketterän projektin onnistumisen kannalta ammattitaitoisten ihmisten sekä ketterää kehittämistä tukevan yrityskulttuurin ohella (Lindvall et al., 2002).

Kevyiden ja ketterien menetelmien tärkeimmiksi puoliksi on mainittu niiden yksinkertaisuus ja nopeus (Abrahamsson, Warsta et al., 2003). Yksittäisen ketterän menetelmän oppiminen vaatii vähemmän tavanomaista koulutusta perinteisiin ohjelmistokehitysprosesseihin verrattuna (Lindvall et al., 2002).

Ketteriä menetelmiä käytetään etenkin ympäristöissä ja tilanteissa, joissa yksiselitteisten ja kattavien vaatimusten tuottaminen on hankalaa, mahdotonta tai jopa asiaankuulumatonta. Vaatimuksia voi joskus olla hankala määritellä tarkasti heti projektin alkaessa, ja ne voivat usein muuttua ja tarkentua projektin edetessä (Cao ja Ramesh, 2008). Ketterät menetelmät pyrkivät reagoimaan nopeasti muutostarpeisiin, jotka voivat liittyä projektin vaatimuksiin, tavoitteisiin, liiketoiminta- tai teknologiaympäristöön (Abrahamsson, Warsta et al., 2003).

Ketteriin menetelmiin liitetään yleensä vahvasti ketterän ohjelmistokehityksen julistus (*Agile Manifesto*) (Agile Alliance, 2001), joka toimii ideologisena pohjana ja yhteisten pelisääntöjen määrittelijänä ketterille menetelmille. Julistus pitää sisällään neljä arvoa ja 12 periaatetta, joiden mukaan ketterät menetelmät toimivat. Julistuksen arvojen mukaisesti ketterässä ohjelmistokehityksessä arvostetaan:

> ***Yksilöitä ja kanssakäymistä*** *enemmän kuin menetelmiä ja työkaluja*
> ***Toimivaa ohjelmistoa*** *enemmän kuin kattavaa dokumentaatiota*
> ***Asiakasyhteistyötä*** *enemmän kuin sopimusneuvotteluja*
> ***Vastaamista muutokseen*** *enemmän kuin pitäytymistä suunnitelmassa*

Julistuksessa vasemmalla puolella esitetyt arvot ovat tärkeimpiä, mutta myös jälkimmäisillä asioilla on arvoa. Ketterässä ohjelmistokehityksessä arvostetaan yksilöitä ja heidän välistä kanssakäymistä. Tämä ilmenee erityisesti läheisen tiimityöskentelyn ja jaettujen työtilojen muodossa. Kehitystiimin yksi tärkeistä tehtävistä on tuottaa toimiva ohjelmisto testien ja useiden pienten julkaisujen avulla. Dokumentoinnin tarve myös vähenee, kun koodi pyritään pitämään mahdollisimman yksinkertaisena, mutta laadukkaana. Kehitystiimin ja asiakkaan välinen yhteistyö ja kommunikaatio nähdään tiukkoja sopimuksia tärkeämmäksi. Liiketoiminnan kannalta ketterillä menetelmillä pyritään tuottamaan asiakkaalle arvoa heti projektin aloittamisen jälkeen, jolloin sopimuksiin liittyvät riskit pienenevät. Ketterän ohjelmistokehitykseen arvoihin kuuluu kyky reagoida muutoksiin, joten asiakas ja kehitystiimi voivat tehdä ohjelmistokehityksen aikana tarvittavia muutoksia, niin ohjelmiston kuin itse prosessinkin kannalta (Abrahamsson, Salo et al., 2002)

Mikään yksittäinen menetelmä ei pysty ratkaisemaan kaikkien erilaisten projektien ongelmia ja vastaamaan niiden tarpeisiin (Abrahamsson, Salo et al., 2002). Projektinhallinnan näkökulmasta tulisi huomioida aina kyseisen projektin tarpeet ja pyrkiä valitsemaan juuri siihen mahdollisimman hyvin soveltuva prosessimalli. Ketterille ja perinteisille suoraviivaisille menetelmille on olemassa sopivia käyttökohteita, joihin ne soveltuvat, joten ketterät menetelmät eivät ole itsessään ole ratkaisu kaikkiin ohjelmistokehityksen ongelmiin (Abrahamsson, Salo et al., 2002; Ramesh et al., 2010; Cao ja Ramesh, 2008).

Ketterien ja perinteisten menetelmien eroja voidaan vertailla tarkastelemalla, miten ne suhtautuvat ohjelmistoprojektin eri osa-alueisiin sekä minkä tyyppisissä projekteissa ne ovat parhaimmillaan ja tuottavat hyviä tuloksia. Tämä voidaan tehdä tarkastelemalla niitä neljällä eri osa-alueella: soveltamiskohteen, hallinnoinnin, teknisen sekä henkilöstön näkökulmasta. Taulukossa 4.1 on esitetty vertailu ketterien ja prosessiorientoituneiden suun-

nitelmavetoisten menetelmien eroista perustuen B. Boehm ja Turner (2003) tekemään kyseisten menetelmien vertailuun. Prosessien soveltamisympäristöissä voidaan havaita eroja siinä, mitkä ovat ketterän ja suunnitelmavetoisen prosessin päätavoitteet, minkä kokoisiin projekteihin ne sopivat parhaiten tiimiin ja tuotettavan ohjelmiston koon suhteen sekä minkälaisessa toimintaympäristössä prosessia suoritetaan. Hallinnon osa-alueella voidaan havaita eroja asiakassuhteiden laadussa, prosessin suhtautumisessa suunnitteluun ja projektin ohjaukseen sekä siinä, miten tietoa kommunikoidaan eri osapuolten välillä. Teknisellä osa-alueella voidaan nähdä eroja menetelmien lähestymistavassa vaatimusten määrittelyyn, ohjelmistokehitykseen ja testaukseen. Henkilöstöön liittyvällä osa-alueella voidaan havaita eroja asiakkaissa, kehittäjissä sekä organisaation kulttuurissa.

Osa-alue	Ketterä	Suunnitelmavetoinen
Soveltamiskohde		
Prosessin päätavoitteet	Tuottaa arvoa heti projektin alusta alkaen ja reagoida nopeasti muutoksiin vaatimuksissa ja mukauttaa prosessia.	Prosessi on ennustettava ja vakaa. Tarkalla prosessin noudattamisella voidaan täyttää vaaditut standardit.
Projektin koko	Pieni tiimi ja projekti	Suuri tiimi ja projekti
Projektin ympäristö	Epävakaa ympäristö, paljon muutoksia, fokus projektissa	Vakaa ympäristö, vähän muutoksia, huomioi projektin ja organisaation tarpeet
Hallinnointi		
Asiakassuhteet	Projektille varattu läsnä oleva asiakas, joka keskittyy projektin priorisoituihin inkrementteihin.	Vuorovaikutusta asiakkaan kanssa tarvittaessa, keskittyy sopimusehtojen määrittämiseen
Suunnittelu ja ohjaus	Sisäistetyt suunnitelmat, jotka kehittyvät ja ohjaavat työtä.	Etukäteen dokumentoidut suunnitelmat, joiden toteutumista voidaan tarkasti seurata.
Kommunikaatio	Tieto siirtyy keskustelemalla.	Tieto liikkuu tuotetun dokumentaation välityksellä.
Tekninen		
Vaatimukset	Vaatimukset ovat priorisoituja epäformaalisti kuvattuja käyttäjätarinoita ja testitapauksia, jotka saattavat muuttua prosessin aikana.	Vaatimukset ovat tarkasti kuvatuttuja, kattavia, jäljitettäviä ja verifioitavia.
Kehitys	Yksinkertainen suunnittelu, lyhyet kehityssyklit, refaktorointi oletettu halvaksi	Kattavat suunnitelmat, pitkät kehityssyklit, refaktorointi oletettu kalliiksi
Testaus	Suoritettavat testitapaukset määrittelevät vaatimuksia.	Sisältää tarkat dokumentoidut testaussuunnitelmat ja menettelytavat.
Henkilöstö		
Asiakkaat	Asiakkaat ovat projektille varattuja, läsnä olevia, yhteistyöhaluisia, edustavia, auktorisoituja, sitoutuneita ja perillä asioista.	Yhteistyöhaluiset, edustavat, auktorisoidut, sitoutuneet ja asioista perillä olevat asiakkaat eivät ole aina läsnä.
Kehittäjät	Prosessin hallitsevia kehittäjiä tarvitaan läpi koko projektin.	Vain projektin alussa on suuri tarve prosessin hallitseville kehittäjille.
Organisaation kulttuuri	Ihmiset ovat vapaita työskentelemään parhaaksi katsomallaan tavalla.	Tarkat säännöt ja käytänteet ohjaavat työelämää.

Taulukko 4.1: Ketterän ja perinteisten lähestymistavan erot ns. kotikenttäetua vertailemalla (B. Boehm ja Turner, 2003)

4.2 Ketteriä prosessimalleja vaatimusmäärittelyn näkökulmasta

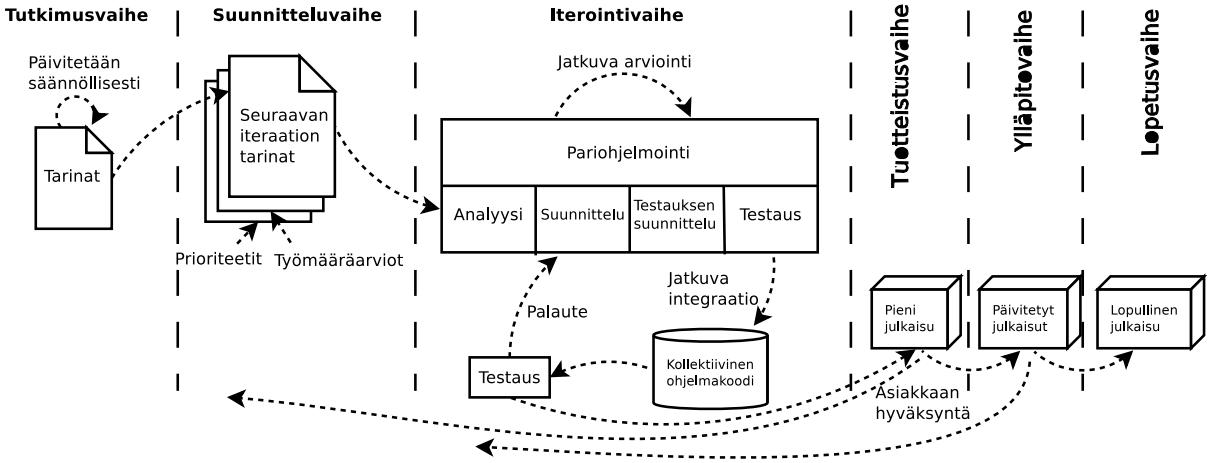
Ketteriä prosessimalleja on tarjolla useita erilaisia. Tässä kappaleessa esitellään niistä erikseen XP ja Scrum, jotka ovat johdannaisineen hyvin suosittuja menetelmiä ohjelmistoteollisuudessa. Ketterään vaatimusmäärittelyyn liittyvässä tutkimuksessa nämä kaksi ovat myös yleensä erikseen mainittuja menetelmiä (Heikkilä et al., 2015). XP on kokoelma hyväksi todettuja käytäntöjä, joilla mahdollistetaan ohjelmistokehitys epämääräisiä tai nopeasti muuttuvia vaatimuksia kohtaaville kehitystiimeille (Beck, 2000). Scrum on iteratiivinen, inkrementaalinen ja mukautuva prosessimalli, jolla hallitaan varsinaista järjestelmän kehitysprosessia varautumalla ja reagoimalla mahdollisiin muutostarpeisiin ohjelmistossa ja itse prosessissa (Sutherland ja Schwaber, 2012).

4.2.1 Extreme programming

XP on eräs tunnetuimmista ketterän ohjelmistokehityksen menetelmistä. Sitä on tutkittu paljon ketterään ohjelmistokehitykseen liittyen (Abrahamsson, Salo et al., 2002) sekä myös ketterään vaatimusmäärittelyyn liittyvässä tutkimuksessa se on ollut usein yksi erikseen mainituista ketteristä menetelmistä (Heikkilä et al., 2015).

XP eli *Extreme Programming* (Beck, 2000) kehitettiin vastaamaan ongelmiin, jotka liittyvät perinteisten menetelmien pitkiin kehityssykleihin. XP on todettu alun perin sopivaksi pienille tai keskisuurille kehitystiimeille, jotka kohtaavat epämääräisiä tai nopeasti muuttuvia vaatimuksia. XP:n sisältämät käytänteet ovat kokoelma jo aiemmin tieteellisesti hyväksi todettuja käytänteitä, jotka XP:ssä on koottu yhteen muodostaen uuden ketterän menetelmän. XP:n ideologia ja nimi juontavat juurensa näiden hyvien käytänteiden viemisestä äärimmäisyyksiin. Esimerkiksi jos yksikkötestit tai koodikatselmoinnit koetaan hyviksi tekniikoiksi, niin niitä tulee silloin tehdä aina eikä vain satunnaisesti (Beck, 2000).

XP:n prosessin elinkaari voidaan jakaa kuuteen eri vaiheeseen (Beck, 2000): tutkimus-, suunnittelu-, iteraatio-, tuotteistamis-, ylläpito- ja lopetusvaiheeseen. Prosessi on esitetty myös kuvassa 4.1.



Kuva 4.1: XP-prosessin elinkaari (Abrahamsson, Salo et al., 2002)

Tutkimusvaiheessa määritetään projektin ensimmäisen julkaisun vaatimukset asiakkaan toimesta. Vaatimukset kirjataan yksitellen tarinoiden muodossa kortille, joiden on tarkoitus kuvata ohjelmistoon lisättäviä ominaisuuksia. Kehitystiimi tutustuu samaan aikaan projektissa käytettäviin teknologioihin, työkaluihin ja käytänteisiin. Vaiheessa voidaan rakentaa järjestelmän prototyyppi arkkitehtuuri- ja teknologiavaihtoehtojen kartoittamiseksi. Kehitystiimin kokemus käytettävistä teknologioista vaikuttaa vaiheen kestoon. Tutkimusvaihe kestää tyypillisesti muutamasta viikosta muutamaan kuukauteen (Abrahamsson, Salo et al., 2002).

Suunnitteluvaiheessa ensimmäisen pienen julkaisun sisältö suunnitellaan priorisoitujen vaatimusten pohjalta. Ensimmäisen julkaisun aikataulu sovitaan kehitystiimin antamaan työmääräarvioon perustuen ja se ei yleensä ylitä kahta kuukautta. Suunnitteluvaihe kestää yleensä pari päivää (Abrahamsson, Salo et al., 2002).

Iterointivaiheessa järjestelmä rakennetaan useassa iteraatiossa siten, että viimeisen iteraation lopussa järjestelmä on valmis tuotantoon. Jokaisen iteraation lopuksi toimitetaan asiakkaalle tietyt ominaisuudet sisältävä toimiva versio. Edellisessä suunnitteluvaiheessa päätetty ensimmäisen julkaisun aikataulu jaetaan useampaan yhdestä viikosta neljään viikkoon kestävään lyhyempään iteraatioon. Ensimmäiseen iteraatioon valitaan sellaiset tarinat, joiden kautta voidaan rakentaa tuotettavan järjestelmän arkkitehtuuri. Asiakkaan vastuulla on valita jokaisen iteraation aikana toteutettavat tarinat ja kirjoittaa jokaisen iteraation päätteeksi ajettavat toiminnalliset testit (Abrahamsson, Salo et al., 2002; Beck, 2000).

Tuotteistusvaiheessa järjestelmä tarkistetaan ja testataan kattavasti ennen sen luovut-

tamista asiakkaalle. Vaiheessa palautesykliä pienennetään, joka voi tarkoittaa iteraation keston lyhentämistä. Uusia vaatimuksia voidaan löytää vielä vaiheen aikana ja niiden osalta päätetään, sisällytetäänkö ne nykyiseen julkaisuun vai toteutetaanko ne myöhemmin seuraavassa julkaisussa esimerkiksi ylläpitovaiheen aikana. Vaiheen aikana pyritään varmistamaan, että kehitettävä ohjelmisto on todella valmis tuotantoa varten (Beck, 2000).

Ylläpitovaiheessa tuotteistettua asiakkaalle toimitettua järjestelmää ylläpidetään tarjoten samalla asiakastukea. Vaiheessa kehitetään myös uusia ominaisuuksia jo olemassa olevaan tuotantojärjestelmään. Iteraatioiden kehitystahti voi tarjottavasta tuesta johtuen hidastua. Ylläpitovaiheen aikana on myös yleistä, että kehittäjätiimin koostumus vaihtelee (Beck, 2000).

Prosessin *lopetusvaiheessa* projektin aktiivinen ylläpito ja kehitystyö lopetetaan. Se voi johtua siitä, että asiakas ja kehittäjät eivät enää pystyä muodostamaan sellaisia uusia ominaisuuksia, joita järjestelmään voitaisiin lisätä tai järjestelmää ei muilta osin tarvitse parantaa (esimerkiksi suorituskyky). Järjestelmä tarpeellinen dokumentaatio kirjoitetaan, sillä järjestelmään ei enää tehdä muutoksia. Projekti voi myös päättyä siihen, että toimitettu ohjelmisto ei enää tuota asiakkaalle sellaista hyötyä, että sen käyttämistä olisi järkevää jatkaa tai jatkokehityksen kustannukset voivat olla liian suuret (Beck, 2000).

XP:hen kuuluu suuri määrä erilaisia siinä hyödynnettäviä käytänteitä (Abrahamsson, Salo et al., 2002; Beck, 2000):

- **Suunnittelupeli** (planning game)
  Suunnittelupelissä kehittäjät arvioivat asiakkaan tuottamien tarinoiden työmäärää, ja asiakas päättää tämän arvion perusteella julkaisujen laajuuden ja ajoituksen.

- **Pienet julkaisut** (small releases)
  Tuotantoversio ohjelmistosta julkaistaan 2–3 kuukauden välein. Uusia toimivia versioita järjestelmästä julkaistaan tämän jälkeen tiheämmin, jopa päivittäin.

- **Metafora** (metaphor)
  Metaforilla kuvataan asiakkaan ja kehitystiimin yhteinen käsitys järjestelmän toiminnasta. Metaforien avulla kaikki projektiin osallistuvat henkilöt voivat ymmärtää perustekijät ja niiden suhteet toisiinsa.

- **Yksinkertainen suunnittelu** (simple design)
  Järjestelmän suunnittelu tulisi pitää niin yksinkertaisena kuin mahdollista. Ylimääräinen monimutkaisuus tulee poistaa heti kun sellaista löydetään.

- **Testaus** (testing)
  Kehitys on testilähtöistä. Yksikkötestit kirjoitetaan ennen varsinaista ohjelmakoodia ja niitä ajetaan jatkuvasti. Asiakkaat kirjoittavat toiminnalliset testit.

- **Refaktorointi** (refactoring)
  Järjestelmän rakennetta parannetaan muuttamatta sen ulkoista toimintaa. Refaktoroinnilla vähennetään koodin toisteisuutta, yksinkertaistetaan ohjelman toimintaa ja lisätään järjestelmän joustavuutta.

- **Pariohjelmointi** (pair programming)
  Kaksi kehittäjää tuottaa ohjelmakoodia samanaikaisesti yhdellä koneella.

- **Koodin yhteisomistajuus** (collective ownership)
  Kaikki tiimin jäsenet voivat muuttaa mitä tahansa osaa koodista. Tiimissä kaikki ovat vastuussa koko järjestelmästä.

- **Jatkuva integraatio** (continous integration)
  Uusia osia integroidaan jatkuvasti osaksi järjestelmän koodia aina niiden valmistuessa. Koodimuutosten täytyy läpäistä kaikki testit, jotta ne hyväksytään.

- **40 tunnin työviikko** (40-hour week)
  Viikon aikana tulee tehdä enintään 40 tuntia töitä. Ylityötä ei saa tapahtua kahta viikkoa peräkkäin. Ylityö on merkki vakavasta ongelmasta projektissa.

- **Asiakkaan läsnäolo** (on-site customer)
  Asiakkaan täytyy olla läsnä ja tiimin käytettävissä koko projektin aika täyspäiväisesti.

- **Koodausstandardit** (coding standards)
  Kehittäjät kirjoittavat koodin tiettyjen konventioiden mukaan, jolloin tuotettu koodi on yhdenmukaista. Yhdenmukaisen koodin kautta kehittäjät ymmärtävät toistensa tuotoksia ja pystyvät kommunikoimaan koodin kautta.

- **Avoin työtila** (open workspace)
  Yhteinen avoin työtila mahdollistaa pariohjelmoinnin ja kommunikaation paremmin.

- **Vain säännöt** (just rules)
  Tiimillä on omat säännöt, joiden mukaan he toimivat. Sääntöjä voidaan muuttaa milloin vain, kunhan tiimi arvioi ja hyväksyy muutokset.

Oleellisimpia näistä XP:n käytänteistä ovat lyhyet iteraation pienillä julkaisuilla, asiakkaan osallistuminen, jatkuva integraatio ja testaus, koodin yhteisomistajuus, vähäinen dokumentaatio sekä pariohjelmointi (Abrahamsson, Salo et al., 2002). Hyödyntämällä useampia XP:n käytänteitä samanaikaisesti voidaan yksittäisen käytänteen heikkoudet ja mahdolliset negatiiviset vaikutukset minimoida. Esimerkiksi ohjelmakoodin refaktorointi saattaisi olla erittäin työlästä ja kallista, ellei samaan aikaan käytettäisi tämän tekniikan heikkouksia pienentäviä käytänteitä, kuten yksikkötestejä ja jatkuvaa integraatiota (Beck, 2000). Toisaalta myös muilla tekijöillä on vaikutuksia käytänteiden tehokkuuteen. Ongelman monimutkaisuus ja kehittäjien kompetenssi voivat esimerkiksi vaikuttaa pariohjelmoinnin tehokkuuteen ja hyödyllisyyteen (Hannay et al., 2009).

XP:ssä keskitytään ohjelmiston rakentamiseen ja varsinaiseen vaatimusmäärittelyyn annetaan hyvin vähän ohjeita. Läsnä oleva asiakas ja kehitystiimi määrittelevät projektin vaatimukset. Pääasiallinen vaatimusmäärittelyn käytänne XP:ssä on siinä tehtävä suunnittelupeli, jossa asiakas vaatimusten kirjoittamisen jälkeen päättää tiimin kanssa seuraavan iteraation aikana toteutettavat vaatimukset. Asiakas myös validoi toteutetut ominaisuudet (Heikkilä et al., 2015). Monet XP-prosessiin liittyvistä ominaisuuksista kuitenkin tukevat vaatimusmäärittelyyn liittyvien vaiheiden suorittamista. Taulussa 4.2 on Lucia ja Qusef (2010) esittämä näkemys vaatimusmäärittelyn vaiheiden toteutumisesta XP:ssä.

Vaatimusmäärittelyn vaihe	**XP**
Esillesaanti	- Vaatimusten määrittely tarinoina - Asiakas kirjoittaa tarinat
Analyysi ja neuvottelu	- Ei erillinen vaihe - Analyysi tapahtuu kehityksen aikana - Asiakas priorisoi käyttäjätarinat
Dokumentointi	- Käyttäjätarinat ja hyväksymistestit vaatimusdokumentaationa - Ohjelmisto pysyvänä informaationa - Kasvokkain tapahtuva kommunikaatio
Validointi	- Testivetoinen kehitys (TDD) - Hyväksymistestien suorittaminen - Palautetta usein
Hallinta	- Lyhyt suunnitteluiteraatio - Käyttäjätarinat tukevat jäljittämistä - Refaktorointi tarvittaessa

Taulukko 4.2: Ketterän vaatimusmäärittelyn toteutuminen XP:ssä (Lucia ja Qusef, 2010)

XP:ssä määritellään eri rooleja projektiin osallistuville ihmisille (Beck, 2000). Nämä ovat: ohjelmoija, asiakas, testaaja, jäljittäjä (*tracker*), valmentaja (*coach*), konsultti ja johtaja. Yksittäisellä henkilöllä voi olla useita rooleja. *Ohjelmoija* kirjoittaa ohjelmakoodia ja testejä pitäen koodin mahdollisimman siistinä ja yksinkertaisena. Hän kommunikoi muiden kanssa selvittääkseen, miten hän tämän tekee. *Asiakas* määrittelee projektin vaatimukset, päättää niiden toteutusjärjestyksen sekä validoi ne. Asiakkaan vastuulla on myös kirjoittaa järjestelmälle toiminnalliset testit, mahdollisesti tiimin avustamana. *Testaajan* tehtävänä on auttaa asiakasta kirjoittamaan toiminnallisia testejä järjestelmälle. Hän ajaa testejä säännöllisesti, raportoi niiden tuloksista ja ylläpitää testaustyökaluja. *Jäljittäjä* seuraa tiimin tekemien työ- ja aikatauluarvioiden toteutumista ja antaa tiimille palautetta niiden tarkkuudesta. Jäljittäjä seuraa jokaisen iteraation päämäärän toteutumista ja antaa arvion seuraavan iteraation mahdollisesta työmäärästä. *Valmentaja* on vastuussa prosessista kokonaisuudessaan. Hän auttaa tiimiä XP:n käytänteiden noudattamisessa ja tarvittaessa ottaa myös enemmän vastuuta projektin ohjaamisesta, jotta asiat saadaan kulkemaan oikeaan suuntaan. *Konsultti* on erityisosaamista omaava tiimin ulkopuolinen jäsen, joka tarvittaessa avustaa kehitystiimiä ongelmien ratkaisemisessa. *Johtaja* vastaa tiimin käytössä olevista resursseista ja vastaa päätösten tekemisestä.

4.2.2 Scrum

Scrum on XP:n ohella yksi hyvin suosituista ketterien kehitysmenetelmien malleista (Heikkilä et al., 2015). Scrum on iteratiivinen, inkrementaalinen ja mukautuva prosessimalli ohjelmistokehitysprojekteille (Sutherland ja Schwaber, 2012). Sen avulla hallitaan varsinaista järjestelmän kehitysprosessia. Prosessin avulla pyritään tuottamaan ohjelmisto varautumalla mahdollisiin muutostarpeisiin ohjelmistossa ja itse prosessissa. Muuttuviin ohjelmiston vaatimuksiin ja toimintaympäristöön voidaan reagoida nopeasti. Prosessissa esiintyviä ongelmia pyritään havaitsemaan ja ratkaisemaan (Abrahamsson, Salo et al., 2002).

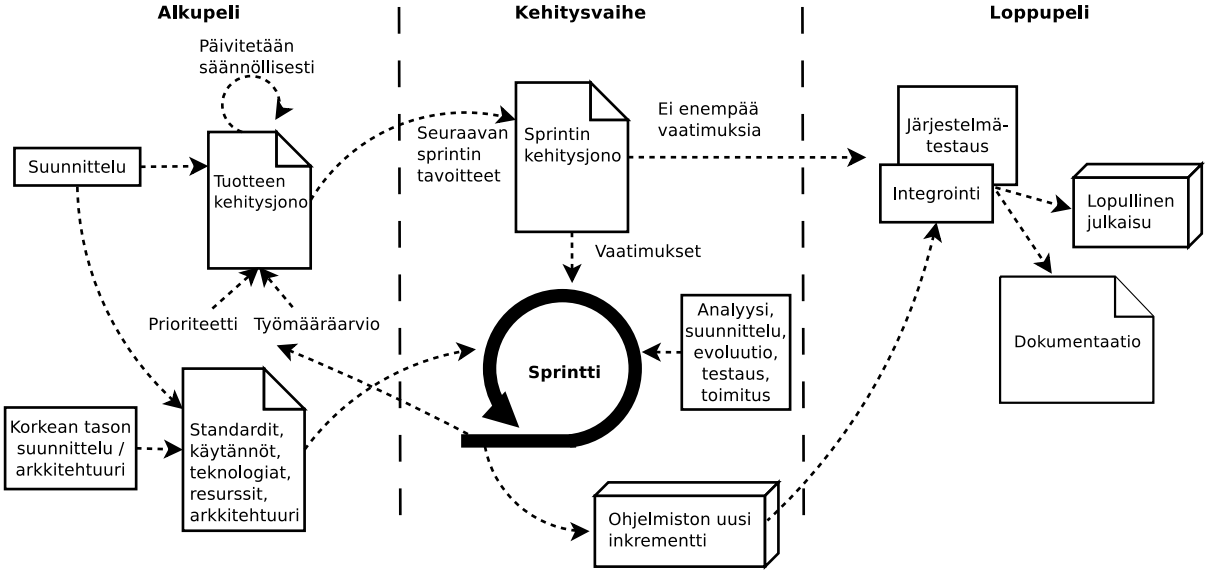
Scrum-prosessi koostuu kolmesta eri vaiheesta: alkupelistä, kehitysvaiheesta ja loppupelistä (Sutherland ja Schwaber, 2012). Prosessi on esitetty kuvassa 4.1.

Alkupeli koostuu kahdesta alavaiheesta: suunnittelusta sekä arkkitehtuurin ja ohjelmiston mallintamisesta korkealla tasolla. Suunnitteluvaiheen aikana kaikki tiedossa olevat ohjelmiston vaatimukset kerätään tuotteen kehitysjonoon. Niille arvioidaan työmäärä ja asetetaan prioriteetti, joita molempia päivitetään jatkuvasti. Myös uusia tai tarkennettuja vaatimuksia lisätään jatkuvasti kehitysjonoon. Vaatimukset voivat tulla useasta eri läh-

teestä. Suunnitteluvaiheessa määritellään myös projektitiimi, käytettävät työkalut ja muut resurssit, riskit, kouluttautumistarpeet sekä varmistetaan johdon hyväksyntä projektille. Arkkitehtuurivaiheessa järjestelmä sekä arkkitehtuuri suunnitellaan korkealla tasolla kehitysjonon sisällön perusteella (Abrahamsson, Salo et al., 2002).

Kehitysvaiheessa järjestelmää kehitetään sprinteissä, jotka ovat iteratiivisia ja inkrementaalisia kehityssyklejä, joiden päätteeksi julkaistaan aina uusi versio kehitettävästä järjestelmästä. Kehitysvaiheen aikana on tarkoitus tuottaa julkaisukelpoinen ohjelmisto, joten vaihe voi sisältää useamman sprintin. Yksittäinen sprintti kestää tyypillisesti yhdestä viikosta kuukauteen. Kehitysvaihe on Scrumin ketterä vaihe, jonka aikana oletetaan ilmenevän erilaisia muutostarpeita. Vaiheen aikana pyritään Scrumin käytänteiden avulla hallitusti reagoimaan ja vastaamaan kaikenlaisiin projektin muutostarpeisiin. Muutoksia voivat aiheuttaa esimerkiksi aikataulut, vaatimukset, resurssit ja käytettävät työkalut (Abrahamsson, Salo et al., 2002).

Loppupelivaiheeseen siirrytään, kun kaikki ohjelmiston julkaisuun liittyvät vaatimukset ja muut tavoitteet on täytetty, eikä uusia ominaisuuksia tai korjauksia ole tarvetta tehdä. Järjestelmä valmistellaan julkaisua varten sekä suoritetaan integrointi, järjestelmätestaus ja kirjoitetaan tarvittava dokumentaatio (Abrahamsson, Salo et al., 2002).



Kuva 4.2: Scrum-prosessin elinkaari (Abrahamsson, Salo et al., 2002)

Scrum ei itsessään sisällä tai vaadi käyttämään mitään tiettyjä ohjelmistokehitysmenetelmiä. Se jättää varsinaiseen ohjelmistokehitykseen liittyvien implementaatiomenetelmien

valitsemisen kehitystiimin vastuulle eikä määrittele niitä, kuten esimerkiksi XP määrittelee muun muassa testauksen ja refaktoroinnin (Abrahamsson, Warsta et al., 2003). Projektia hallitaan ja ohjataan tiettyjen hallinnollisten käytänteiden avulla. Seuraavia käytänteitä hyödynnetään Scrumissa projektinhallintaan (Abrahamsson, Salo et al., 2002; Sutherland ja Schwaber, 2012; Schwaber ja Sutherland, 2017):

- **Tuotteen kehitysjono** (product backlog)
  Tuotteen kehitysjono sisältää kaikki projektissa tehtävät työt. Se sisältää jatkuvasti päivitettävän priorisoidun listan rakennettavan järjestelmän teknisistä ja muista vaatimuksista. Tehtävät ovat luonteeltaan ominaisuuksia, bugikorjauksia, virheitä, parannuksia ja käytettävän teknologian päivityksiä. Tehtävät voivat tulla useasta eri lähteestä, kuten asiakkaalta tai kehitystiimiltä. Tuoteomistaja on vastuussa kehitysjonon ylläpidosta eli kehitysjonon luomisesta sekä tehtävien lisäämisestä, poistamisesta, päivittämisestä ja priorisoimisesta.

- **Työmääräarviot** (effort estimation)
  Työmääräarvioita tehdään iteratiivisesti päivittäen niitä, kun uutta tietoa saadaan kehitysjonossa oleviin tehtäviin. Työmääräarviot tehdään tuoteomistajan ja kehitystiimin yhteystyönä.

- **Sprintti** (sprint)
  Sprintin aikana Scrum-tiimi tuottaa uuden toimivan version kehitettävästä ohjelmasta. Yksittäisen sprintin pituus on viikosta kuukauteen ja kaikki sprintit ovat yleensä keskenään saman pituisia. Sprintin aikana hyödynnettävät muut Scrumin käytänteet ovat sprintin suunnittelu, sprintin kehitysjono, päivittäispalaverit, edistymiskäyrät sekä tuotteen kehitysjonon jalostaminen.

- **Sprintin suunnittelu** (sprint planning)
  Sprintin suunnittelussa määritellään tulevan sprintin tavoitteet ja valitaan tuotteen kehitysjonosta sprintin aikana toteutettavat tehtävät asiakkaan, käyttäjien, johdon, tuoteomistajan ja kehitystiimin toimesta. Palaverin lopuksi Scrum-mestari yhdessä tiimin kanssa selvittää, miten ohjelmiston inkrementti sprintin aikana toteutetaan. Scrum-mestari vastaa palaverin järjestämisestä.

- **Sprintin kehitysjono** (sprint backlog)
  Sprintin kehitysjono pitää sisällään sprintin aikana toteutettavaksi valitut tehtävät tuotteen kehitysjonosta. Kehitysjono muodostetaan sprintin suunnittelussa kehitys-

tiimin, Scrum-mestarin ja tuoteomistajan toimesta sprintin tavoitteiden ja prioriteettien perusteella. Sprintin kehitysjono pysyy vakaana sprintin päättymiseen saakka, toisin kuin tuotteen kehitysjono, jota voidaan päivittää ja priorisoida jatkuvasti. Uusi versio ohjelmistosta toimitetaan sprintin kehitysjonon tehtävien valmistuttua.

- **Päivittäispalaveri** (daily scrum)
  Päivittäisessä palaverissa seurataan sprintin työn edistymistä. Mahdolliset työtä estävät tai hidastavat ongelmat nostetaan esille ja pyritään ratkaisemaan. Scrum-tiimi myös suunnittelee työtä käymällä läpi, mitä edellisen palaverin jälkeen on saavutettu ja mitä aiotaan saavuttaa seuraavaan mennessä. Palaveri on tarkoitus pitää hyvin lyhyenä (noin 15 minuuttia).

- **Edistymiskäyrä** (burndown chart)
  Edistymiskäyrän avulla seurataan työn etenemistä sprintin aikana. Kehitystiimin täytyy pitää sprintin tehtävien jäljellä oleva työmäärä ajan tasalla.

- **Tuotteen kehitysjonon jalostaminen** (product backlog refinement)
  Jalostamisen tarkoitus on parannella tuotteen kehitysjonossa olevia nykyiseen sprinttiin kuulumattomia tehtäviä, jotta tulevien sprinttien kehitysjonon sisällön suunnittelu helpottuisi. Jalostamiseen voidaan tyypillisesti varata muutama tunti jokaisessa sprintissä. Jalostamisella voidaan tarkentaa kehitysjonossa olevia tehtäviä, pilkkoa liian suuria kokonaisuuksia ja tehdä ja tarkentaa työmääräarvioita kehitysjonon tehtäville.

- **Sprintin katselmointi** (sprint review)
  Sprintin viimeisenä päivänä kehitystiimi esittelee Scrum-mestarin kanssa sprintin tulokset asiakkaalle, johdolle, käyttäjille sekä tuoteomistajalle. Katselmointiin osallistuvat arvioivat tuloksen ja päättävät jatkotoimista. Katselmoinnin seurauksena kehitysjono ja projektin suunta saattavat muuttua.

- **Inkrementti** (increment)
  Inkrementti on uusi versio ohjelmistosta, joka täyttää Scrum-tiimin määritelmän valmiista ja on käyttökelpoisessa kunnossa. Se pitää sisällään kaikki kehitysjonon kohdat, jotka ovat valmistuneet nykyisen ja aiempien sprinttien aikana.

- **Sprintin retrospektiivi** (sprint retrospective)
  Retrospektiivi pidetään sprintin katselmoinnin jälkeen ennen uuden sprintin aloittamista. Kehitystiimi ja Scrum-mestari tarkastelevat siinä itse prosessia ohjelmiston

sijaan. He pyrkivät tunnistamaan ja tekemään prosessiin tarvittavia parannuksia, jotta seuraava sprintti sujuisi edellisiä paremmin.

Scrum määrittelee projektiin osallistuville osapuolille kolme erilaista roolia: tuoteomistaja, Scrum-mestari ja kehitystiimi. Nämä yhdessä muodostavat Scrum-tiimin (Sutherland ja Schwaber, 2012). Projektiin voi kuulua myös joukko avustavia rooleja, kuten asiakas ja johtoryhmä.

Scrum-mestarin tehtävä on varmistaa ja pitää huolta Scrum-prosessin noudattamisesta ja varmistaa projektin eteneminen suunnitelmien mukaisesti. Hän myös pyrkii parantamaan prosessia tiimin kanssa, jotta työ voidaan suorittaa mahdollisimman hyvin. *Tuoteomistaja* on vastuussa projektista ja tuotteen kehitysjonosta. Hän edustaa eri sidosryhmiä ja toimii linkkinä heidän ja kehitystiimin välillä. Itseorganisoituva *kehitystiimi* vastaa sprintin kehitysjonon tehtävien ja tavoitteiden toteutuksesta itsenäisesti. *Asiakas* osallistuu tuotteen kehitysjonoon liittyviin tapahtumiin. *Johtoryhmän* vastuulla ovat päätökset projektin hyväksymisestä, vaatimuksista ja tavoitteista (Abrahamsson, Salo et al., 2002).

Scrum sisältää vaatimusmäärittelyä tukevia käytänteitä ja tietyt roolit ovat niiden määrittelystä erityisen suuressa vastuussa. Taulussa 4.3 on listattu Lucia ja Qusef (2010) esittämä näkemys eri vaatimusmäärittelyn vaiheiden toteutumisesta Scrumissa. Scrumissa tuoteomistajalla on suurin vastuu vaatimusten esillesaannista ja priorisoinnista. Vaatimukset kirjataan tuotteen kehitysjonoon, joka on priorisoitu lista kaikista ohjelmistoon liittyvistä työtehtävistä. Tuotteen kehitysjono on tuoteomistajan vastuulla ja ainoastaan hänellä on oikeus hallita sen sisältöä ja lisätä uusia tehtäviä. Tuoteomistaja yhdessä kehitystiimin kanssa epämuodollisesti analysoi ja validoi vaatimuksia sekä päättää seuraavan sprintin aikana toteutettavat työtehtävät kuullen muita sidosryhmiä. Sprintin päätteeksi sen aikana valmistuneet vaatimukset validoidaan tuoteomistajan ja muiden sidosryhmien toimesta (Heikkilä et al., 2015).

Vaatimusmäärittelyn vaihe	**Scrum**
Esillesaanti	- Tuoteomistaja luo ja hallitsee tuotteen kehitysjonoa - Sidosryhmät voivat osallistua tuotteen kehitysjonon määrittämiseen
Analyysi ja neuvottelu	- Tuotteen kehitysjonon jalostaminen - Tuoteomistaja priorisoi tuotteen kehitysjonoa - Tuoteomistaja analysoi vaatimusten toteuttamiskelpoisuutta
Dokumentointi	- Kasvokkain tapahtuva kommunikaatio
Validointi	- Katselmointipalaverit
Hallinta	- Sprintin suunnittelu - Tuotteen kehitysjonon tehtävät tukevat jäljittämistä - Muuttuneet vaatimukset lisätään/poistetaan tuotteen kehitysjonosta

Taulukko 4.3: Ketterän vaatimusmäärittelyn toteutuminen Scrumissa (Lucia ja Qusef, 2010)

4.3 Ketterä vaatimusmäärittely

Tässä luvussa käsitellään ketterässä ohjelmistoprojektissa tapahtuvaa vaatimusmäärittelyä. Ensimmäiseksi luvussa 4.3.1 käydään läpi, miten vaatimusmäärittely tapahtuu ketterissä projekteissa. Tämä tehdään vertailemalla perinteistä ja ketterää vaatimusmäärittelyä perinteisen vaatimusmäärittelyprosessin vaihejaon kautta (katso luku 3.3). Luvussa 4.3.2 tarkastellaan ketterällä vaatimusmäärittelyllä saavutettavissa olevia hyötyjä perinteiseen vaatimusmäärittelyyn verrattuna sekä listataan haasteita, joita voidaan kohdata määriteltäessä ohjelmistoprojektin vaatimuksia ketterästi. Seuraavaksi luvussa 4.3.3 käydään läpi tutkimuskirjallisuudessa tunnistettuja käytänteitä, joilla vaatimusmäärittelyä ketterissä ohjelmistoprojekteissa tehdään. Lopuksi luvussa 4.3.4 tarkastellaan, millaisia riskejä vaatimuksiin liittyy ohjelmistoprojekteissa ja miten ketterän vaatimusmäärittelyn käytänteiden hyödyntäminen vaikuttaa näihin riskeihin.

Ketterillä ja perinteisillä menetelmillä on hyvin erilaiset lähtöoletukset liittyen vaatimusmäärittelyyn. Ketterät menetelmät olettavat (Turk et al., 2005), että:

- Vaatimukset tulevat todennäköisesti muuttumaan monesta syystä, jonka vuoksi kaik-

kien vaatimusten selvittäminen kattavasti etukäteen ei ole kannattavaa.

- Tarkan dokumentaation tuottaminen on hyödytöntä, sillä lopullinen spesifikaatio nähdään tuotetusta koodista eikä dokumentaatiosta.
- Asiakkaat ovat kehitystiimin saatavilla jatkuvasti, jotta vaatimukset voidaan saada selville ja validoida.
- Muutosten tekeminen järjestelmään ei muutu kalliimmaksi ajan kanssa, joten iteratiivisesti ja inkrementaalisesti tehtävä vaatimusmäärittely ei kasvata kehityskustannuksia.

Perinteiset ohjelmistokehitysmenetelmät puolestaan olettavat (Sillitti et al., 2005), että:

- Asiakas on kykenevä ilmaisemaan kaikki järjestelmän vaatimukset etukäteen.
- Yksi tai useampi osapuoli on vastuussa vaatimusten tuottamisesta. Asiakas on harvoin tekemisissä suoraan kehittäjien kanssa.
- Kehitystiimi kykenee helposti ymmärtämään kaikki asiakkaan tarpeet ilman tarvetta selventää asioita myöhemmin.
- Organisaatioiden rakenne on hierarkkinen ja eri toimien välillä on selvät rajat, joka aiheuttaa ristiriitoja vaatimusmäärittelyssä.

Ketterä vaatimusmäärittely ei ole sen merkityksen kannalta täysin tarkkaan määritelty käsite, sillä menetelmät itse eivät sitä eksplisiittisesti määrittele. Ketterään vaatimusmäärittelyyn liittyvä tutkimus on keskittynyt vaatimusmäärittelyyn liittyvien toimintojen tunnistamiseen ja kutsunut näitä ketteräksi vaatimusmäärittelyksi (Heikkilä et al., 2015). Ketterä vaatimusmäärittely ymmärretään tämän tutkielman kannalta vaatimusmäärittelyksi, joka tapahtuu osana ketterää ohjelmistoprojektia. Siinä hyödynnetään sellaisia käytänteitä, jotka tukevat ja toteuttavat vaatimusmäärittelyssä perinteisesti olevia prosessin vaiheita, vaikkakin käytänteet itsessään eroavat perinteisistä.

Asiakkaan ja kehitystiimin välinen jatkuva ja läheinen kommunikaatio on ketterässä vaatimusmäärittelyssä oleellisessa asemassa. Vaatimukset muodostuvat iteratiivisesti jatkuvan asiakkaan ja kehitystiimin välisen kommunikaation kautta (Heikkilä et al., 2015).

4.3.1 Vaatimusmäärittelyprosessin toteutuminen

Vaatimusmäärittelyn toteutumista ketterässä ohjelmistokehityksessä voidaan tarkastella perinteisen vaatimusmäärittelyprosessin vaiheiden kautta. Luvussa 3.3 käytiin läpi vaatimusten esillesaanti, analyysi ja neuvottelu, dokumentointi sekä validointi. Lisäksi erikseen mainittiin myös vaatimusten hallinta. Taulukossa 4.4 esitetään Ramesh et al. (2010) mukainen vertailu vaatimusmäärittelyn vaiheiden toteutumisesta perinteisissä ja ketterissä menetelmissä.

Ketterän vaatimusmäärittelyn avulla voidaan suorittaa kaikkia perinteisen vaatimusmäärittelyn vaiheita. Vaiheiden suorittamiseen käytetään perinteisistä menetelmistä poikkeavia käytänteitä eikä vaiheita suoriteta lineaarisesti peräkkäin vaan iteratiivisesti jokaisen usean lyhyen kehityssyklin aikana. Ketterässä vaatimusmäärittelyssä vaiheiden välillä ei ole selvää rajaa, vaan ne sekoittuvat keskenään erilaisista käytänteistä johtuen (Ramesh et al., 2010; Lucia ja Qusef, 2010; Paetsch et al., 2003).

Asiakkaan ja kehitystiimin välinen suora, epäformaali ja välitön kommunikaatio on ketterän vaatimusmäärittelyn oleellinen piirre. Vaatimukset saadaan esille, analysoidaan, dokumentoidaan ja validoidaan keskustelun kautta (Ramesh et al., 2010).

Vaatimusmäärittelyn vaihe	**Perinteinen vaatimusmäärittely**	**Ketterä vaatimusmäärittely**
Esillesaanti	Kaikki vaatimukset pyritään löytämään projektin alussa ennen kehityksen alkamista. Muutokset ovat hankalia myöhemmin.	Vaatimusmäärittelyä tehdään iteratiivisesti. Vaatimuksia löydetään läpi kehitysprosessin. Käytänteet: iteratiivinen vaatimusmäärittely, kasvokkain käytävä kommunikaatio
Analyysi ja neuvottelu	Keskittyy sidosryhmien ristiriitojen ratkaisemiseen.	Vaatimuksia parannellaan, muutetaan ja priorisoidaan iteratiivisesti. Käytänteet: iteratiivinen vaatimusmäärittely, kasvokkain käytävä kommunikaatio, jatkuva suunnittelu, jatkuva priorisointi
Dokumentointi	Formaali dokumentaatio sisältää tarkat kuvaukset vaatimuksista. Tieto liikkuu dokumenttien avulla.	Ei formaalia dokumentaatiota. Vaatimukset kuvataan vapaamuotoisemmin esimerkiksi käyttäjätarinoina. Tieto välittyy keskustelun avulla. Käytänteet: kasvokkain käytävä kommunikaatio
Validointi	Vaatimusmäärittelydokumentti validoidaan tarkasti ja varmistetaan sen johdonmukaisuus ja kattavuus.	Varmistutaan, että vaatimukset vastaavat nykyisiä käyttäjän tarpeita. Validointi tapahtuu kommunikaation kautta. Käytänteet: katselmointipalaverit, kasvokkain käytävä kommunikaatio

Taulukko 4.4: Vertailu vaatimusmäärittelyn vaiheiden toteutumisesta perinteisissä ja ketterissä menetelmissä (Ramesh et al., 2010)

Ketterässä ohjelmistokehityksessä vaatimusten esillesaanti suoritetaan iteratiivisen vaatimusmäärittelyn ja kasvokkain asiakkaan kanssa tapahtuvan kommunikaation avulla (Ramesh et al., 2010). Uudet tai muuttuneet vaatimukset hyväksytään vielä myöhäisessäkin vaiheessa, kun perinteiset menetelmät puolestaan pyrkivät selvittämään kaikki ohjelmiston vaatimukset ja valmistelemaan vaatimusmäärittelydokumentaation ennen kehitystyön

aloittamista (Lucia ja Qusef, 2010).

Vaatimusten analysointi ja neuvottelu tapahtuvat iteratiivisen vaatimusmäärittelyn, kasvokkain tapahtuvan kommunikaation ja jatkuvan suunnittelun sekä jatkuvan priorisoinnin kautta. Projektin alussa selvitetään karkeasti järjestelmän kriittisimmät vaatimukset, jotka toimivat lähtökohtana kehitykselle ja ensimmäisen kehityssyklin suunnittelulle. Vaatimukset muuttuvat ja niitä löydetään koko kehitysprosessin ajan. Ketterän vaatimusmäärittelyn käytänteet auttava parantamaan, muuttamaan ja priorisoimaan vaatimuksia (Ramesh et al., 2010). Tarvittavien vaatimusten tarkempi määrittely vasta juuri ennen toteutusta voi myös luoda kustannussäästöjä, kun määrittelystä aiheutuva työ voidaan tehdä vasta, kun sitä tarvitaan (Paetsch et al., 2003).

Kattava yksityiskohtainen vaatimusdokumentaatio on ketterässä vaatimusmäärittelyssä korvattu asiakkaan ja kehitystiimin välisellä kommunikaatiolla. Tieto ei liiku dokumentaation välityksellä, vaan kaikki tieto kommunikoidaan sanallisesti eri osapuolten välillä. Vaatimukset voidaan kirjata vapaammin esimerkiksi priorisoituihin listoihin ominaisuuksista tai käyttäjätarinoihin formaalin vaatimusmäärittelydokumentaation sijaan (Ramesh et al., 2010).

Vaatimuksia validoidaan formaalisti katselmointipalaverien aikana, mutta validointia tapahtuu myös epävirallisemmin asiakkaan ja kehitystiimin välisen kommunikaation kautta esimerkiksi suunniteltaessa uutta iteraatiota tai sen aikana. Vaatimusten validoinnissa keskitytään ketterässä vaatimusmäärittelyssä varmistamaan asiakkaan tarpeiden täyttyminen, sillä vaatimusdokumentaation johdonmukaisuutta ja kattavuutta ei muodollisesti varmisteta sen puuttumisen vuoksi (Ramesh et al., 2010).

4.3.2 Hyödyt ja haasteet

Ketterällä vaatimusmäärittelyllä on tutkimuskirjallisuudessa esitetty olevan tiettyjä hyötyjä verrattaessa sitä perinteiseen vaatimusmäärittelyprosessiin. Hyötyjen lisäksi on tutkittu myös ketterään vaatimusmäärittelyyn liittyviä haasteita ja mahdollisia ratkaisuja niihin. Osa mainituista hyödyistä ja haasteista on verrattain samoja kuin ketterällä ohjelmistokehityksellä on yleisesti ottaen esitetty olevan. Heikkilä et al. (2015) kartoittivat artikkelissaan *A Mapping Study on Requirements Engineering in Agile Software Development* ketterään vaatimusmäärittelyyn liittyvää tutkimusta. He listasivat ketterään vaatimusmäärittelyyn liitettäviä hyötyjä ja haasteita sekä mahdollisia ratkaisuja näihin haasteisiin. Seuraavaksi käydään läpi artikkelissa esitetyt kuusi hyötyä ja kuusi haastetta. Tätä

listausta hyödynnetään tapaustutkimuksen analyysissä. Luvussa 6.2 esitetään ketterässä ohjelmistoprojektissa havaitut ketterän vaatimusmäärittelyn hyödyt ja haasteet.

Vähemmän työtä prosessin vuoksi: Prosessin suorittamisesta itsestään koituu vähemmän työtä, ja prosessi on kevyempi sekä vaatii vähemmän tuotoksia. Perinteisesti vaatimusten osalta suoritetaan suuritöinen dokumentointi- ja hyväksymisprosessi, jota ei ketterässä vaatimusmäärittelyssä yleensä tehdä (Paetsch et al., 2003; Cao ja Ramesh, 2008). Vaatimuksiin liittyvät ongelmat voidaan ratkaista jo niistä keskusteltaessa, mistä seuraa pienempi tarve niiden uudelleentyöstämiselle (Bjarnason et al., 2011).

Parantunut ymmärrys vaatimuksista: Hyvällä keskusteluyhteydellä asiakkaan kanssa saavutetaan se, että vaatimukset ja niiden prioriteetit voidaan ymmärtää paremmin. Vaatimukset ovat myös tarvittaessa nopeasti validoitavissa asiakkaan kanssa (Bjarnason et al., 2011; Cao ja Ramesh, 2008; Ramesh et al., 2010).

Vähentynyt ylikuormittaminen: Todennäköisyys sille, että käytettävissä olevat kehitysresurssit ylikuormitetaan, on pienempi. Myös kehitystyön laajuutta on helpompi hallita. Vaatimuksia on mahdollista jatkuvasti priorisoida yhden listan (työjonon) avulla ja toteutettavan työn määrästä sovitaan aina käytettävissä olevan kehityskapasiteettiin mukaan (Bjarnason et al., 2011). Esimerkiksi suunniteltaessa yksittäistä Scrum-sprinttiä, pyritään siihen ottamaan mukaan vain sen verran työtä kuin arvioidaan, että on mahdollista saavuttaa sprintin aikana.

Muutostarpeeseen vastaaminen: Ohjelmiston vaatimusten sisältö, prioriteetti ja määrä reagoivat muutostarpeeseen paremmin ketterällä vaatimusmäärittelyllä. Muutokset ovat helpommin ja halvemmin toteutettavissa. Muutostarpeen voivat laukaista esimerkiksi muutokset toimintaympäristössä tai voi olla, että vaatimukset syntyvät ja tarkentuvat käytön sekä tavoitteiden ymmärryksen kautta eivätkä niinkään etukäteen määriteltynä. Vaatimusten muutoksen salliminen kehityksen aikana mahdollistaa myös sen, että järjestelmä vastaa paremmin asiakkaan tarpeita (Cao ja Ramesh, 2008; Paetsch et al., 2003; B. Boehm, 2000; Ramesh et al., 2010).

Nopeat toimitukset ja validaatio: Ketteriä vaatimusmäärittelyn menetelmiä hyödyntäen saadaan nopeammin tuotettua arvoa sekä validoitua ratkaisuja. Lyhyillä kehitysiteraatioilla voidaan nopeammin täyttää asiakkaan tärkeimpiä vaatimuksia. Katselmointien avulla vaatimukset täyttäviä toimitettuja ratkaisuja voidaan helposti validoida (Cao ja Ramesh, 2008; Paetsch et al., 2003).

Parantuneet asiakassuhteet: Suhteet asiakkaisiin ja käyttäjiin parantuvat ja ovat tyy-

dyttävämpiä käytettäessä ketterää vaatimusmäärittelyä (Cao ja Ramesh, 2008).

Hyöty	**Kuvaus**
Vähemmän työtä prosessin vuoksi	Vähemmän prosessin vuoksi tehtävää työtä, tuotoksia ja hukkaa.
Parantunut ymmärrys vaatimuksista	Suoran kommunikaation kautta eri osapuolten välillä vaatimukset ja niiden prioriteetit voidaan ymmärtää paremmin sekä vaatimukset voidaan validoida nopeasti.
Vähentynyt ylikuormittaminen	Kehitysresursseja ei yritetä käyttää enempää kuin niitä on saatavilla ja kehitystyön laajuutta on helpompi hallita.
Muutostarpeeseen vastaaminen	Vaatimusten sisältö ja prioriteetti pystyvät vastaamaan muutostarpeeseen helpommin ketterällä vaatimusmäärittelyllä.
Nopeat toimitukset ja validaatio	Asiakkaalle saadaan nopeasti tuotettua arvoa sekä voidaan nopeasti validoida vaatimuksia asiakkaan kanssa.
Parantuneet asiakassuhteet	Suhteet asiakkaisiin ja käyttäjiin parantuvat ketterällä vaatimusmäärittelyllä.

Taulukko 4.5: Yhteenveto ketterän vaatimusmäärittelyn esitetyistä hyödyistä (Heikkilä et al., 2015)

Ongelmat liittyen asiakkaaseen tai asiakasedustajiin: Ketterä vaatimusmäärittely vaatii toimiakseen paljon asiakkaan ja kehitystiimin välistä vuorovaikutusta sekä luottamusta. Näiden saavuttaminen voi olla hankalaa, sillä asiakas ei välttämättä ymmärrä ketterää vaatimusmäärittelyprosessia tai luota siihen. Luottamuksen puuttumisella on erityisen suuri vaikutus projektiin. Kehitystiimi tarvitsee suoran yhteyden asiakkaaseen tai osaavaan asiakasedustajaan, joka on myös valmis jatkuvasti validoimaan vaatimuksia ja kehitystiimin tuottamia ratkaisuja. Suora yhteys asiakkaaseen on aina tehokkaampi kuin välikäsien kautta saavutettu tieto, mutta jatkuvasti käytettävissä olevaa asiakasedustusta on hankala saada. Tuoteomistajien kaltaisten asiakasedustajien rooli on tärkeä suurissa, useita asiakkaita käsittävissä projekteissa, joissa heidän on oleellista toimia välikätenä kehitystiimin ja useiden asiakkaiden välillä. Tämänkaltaisten asiakasedustajien puuttuessa vaatimusten arviointi tapahtuu kehittäjien toimesta, joilta voi puuttuu oleellinen ymmärrys todellisista prioriteeteista (Cao ja Ramesh, 2008; Ramesh et al., 2010; Heikkilä et al., 2015). Huonosta vuorovaikutussuhteesta voi seurata puutteellisia, väärin määriteltyjä ja

väärin toteutettuja vaatimuksia. Kehittäjien on myös mahdollisesti neuvoteltava useiden sidosryhmien kesken vaatimuksista (Cao ja Ramesh, 2008).

Perinteisempien vaatimusmäärittelykäytänteiden ja prosessien hyödyntämistä on esitetty ratkaisuksi asiakkaaseen tai asiakasedustajiin liittyviin ongelmiin. Näistä esimerkkejä ovat vaatimusmäärittelijän roolin määrittäminen projektiin, erillisen vaatimusten esillesaantivaiheen suorittaminen ja ylimääräisen vaatimusdokumentaation tuottaminen. Automaattisten testien (esim. *acceptance test-driven development*) hyödyntämistä vaatimusten validoinnissa on myös ehdotettu (Heikkilä et al., 2015).

Käyttäjätarinoiden riittämättömyys: Käyttäjätarinoiden avulla voidaan kuvata ainoastaan yksinkertaisia käyttäjälle näkyviä toiminnallisuuksia. Yksittäinen käyttäjätarina voi vaatia muutoksia useisiin järjestelmän osiin, ja se täytyy mahdollisesti pilkkoa vielä pienempiin vaatimuksiin. Ketterällä vaatimusmäärittelyllä vaatimuksia pystytään validoimaan tehokkaasti esimerkiksi katselmointipalavereissa, mutta tarinoiden keskinäinen yhteensopivuus eli johdonmukaisuus tai verifioitavuus on hankalaa selvittää. Ketterä vaatimusmäärittely ja käyttäjätarinat eivät erikseen tue vaatimusten jäljitettävyyttä eikä vaatimuksiin liittyviä päätöksiä yleensä myöskään dokumentoida, jolloin vaatimusten tai koodin muuttuessa syy-seuraussuhde ei ole helposti nähtävissä. Ei-toiminnalliset vaatimukset jätetään helposti huomiotta tai niiden toteutus perustuu puhtaasti hiljaiseen tietoon. Asiakkaiden huomio on yleensä helposti lähestyttävissä toiminnallisissa vaatimuksissa, kun taas monet ei-toiminnallisista vaatimuksista täytyisi pitää mielessä kaikessa toteutuksessa. Ei-toiminnallisia vaatimuksia on tästä syystä myös haastavaa validoida katselmointien yhteydessä käytettävyyttä lukuun ottamatta. Käyttäjätarinat eivät sisällä tarpeeksi tietoa suurten ja monimutkaisten järjestelmien rakentamiseen vaadittavista suunnitelmista, joten erillisiä järjestelmätason vaatimuksia tarvitaan myös (Cao ja Ramesh, 2008; Paetsch et al., 2003; Savolainen et al., 2010; Ramesh et al., 2010).

Ongelmat käyttäjätarinoissa voivat johtua myös niiden heikosta laadusta. Korkeatasoiset vaatimukset edesauttavat projektin onnistumisessa, joten käyttäjätarinoillekin on ehdotettu malleja niiden laadun parantamiseen (Lucassen et al., 2015). Muita ratkaisuja käyttäjätarinoihin liittyviin ongelmiin voivat olla lisätä tarinaformaatin pakollisen informaation määrää, luoda hierarkiaa tarinoihin ja lisätä niiden jäljitettävyyttä, tai käyttää tarinoita ainoastaan vaatimusten esillesaantiin ja suorittaa vaatimusten tarkempi määrittely perinteisempiä analysointi- ja verifiointitekniikoita käyttäen (Heikkilä et al., 2015).

Haasteet vaatimusten priorisoinnissa: Vaatimukset, jotka tuottavat eniten välitöntä arvoa, priorisoidaan yleensä etusijalle. Tästä voi seurata, että näennäisesti epäolennai-

semmat arkkitehtuuriin, järjestelmän parannuksiin tai joihinkin ei-toiminnallisiin vaatimuksiin liittyvät vaatimukset jäävät aluksi vähemmälle huomiolle. Asiakkaille voi tuottaa vaikeuksia kyvykkyydestä tai halusta riippuen määritellä vaatimusten prioriteettia. Asiakkaan mielestä kaikki vaatimukset voivat olla myös prioriteetiltaan kriittisiä. Eri osapuolten tarpeet järjestelmää kohtaan voivat olla erilaisia, jolloin näkemykset vaatimusten prioriteeteista voivat poiketa ja yhteisymmärryksen saavuttaminen voi olla hankalaa. Nopeasti toimitetut, asiakkaalle näkyvät vaatimukset (esimerkiksi prototyyppien muodossa) voivat aiheuttaa epärealistisia odotuksia asiakkaalle, jolloin lopullisen toteutuksen vaatiman suuremman työmäärän hyväksyminen voi olla haastavaa (Cao ja Ramesh, 2008; Ramesh et al., 2010; Bjarnason et al., 2011).

Kasvava tekninen velka: Ketterän vaatimusmäärittelyn lyhyen tähtäimen suunnittelu voi johtaa teknisiin ratkaisuihin, jotka eivät pysty vastaamaan enää muuttuvien, uusien tai myöhemmin toteutettavien vaatimusten tarpeisiin, jolloin niiden toteuttaminen hankaloituu. Tämä puolestaan voi johtaa arkkitehtuuriin, joka muodostuu riittämättömäksi tai vääränlaiseksi jatkon kannalta. Arkkitehtuurin refaktorointi voi vaatia jopa koko järjestelmän uudelleenkirjoittamisen. Myös refaktoroinnin priorisointi voi olla haastavaa, sillä sen tarkoitus ei ole tuottaa uusia ominaisuuksia, vaan parantaa vain sisäistä toimintaa (Cao ja Ramesh, 2008; Ramesh et al., 2010).

Tukeutuminen hiljaiseen tietoon vaatimuksista: Ketterään vaatimusmäärittelyyn sisältyy paljon hiljaista tietoa. Perinteisten suunnitelmavetoisten ohjelmistoprosessien tuottaman dokumentaation sijaan painoarvo ketterässä vaatimusmäärittelyssä on osaavissa ja pätevissä henkilöissä ja heidän välisessä suorassa kommunikaatiossa. Ketterät menetelmät tuottavat usein dokumentaationaan ainoastaan esimerkiksi käyttäjätarinoita, tuotteen kehitysjonon, sprinttien kehitysjonoja ja edistymiskäyriä. Henkilöstönvaihdokset aiheuttavat haasteita, kun osa tästä kirjaamattomasta tiedosta voidaan menettää (Cao ja Ramesh, 2008; Paetsch et al., 2003). Henkilöstön vaihdoksesta johtuvaan ongelmaa tiedon menettämisen suhteen voidaan yrittää paikata tuottamalla lisää vaatimuksiin liittyvää dokumentaatiota (Paetsch et al., 2003).

Epätarkat työmääräarviot: Tarkkojen työmäärään ja kustannuksiin liittyvien arvioiden antaminen asiakkaalle tai yrityksen johdolle on haasteellista ilman kattavaa vaatimusten analysointia ja spesifiointia. Alustavat arviot saattavat perustua vain tiedossa oleviin käyttötapauksiin, jotka joudutaan arvioimaan melko karkeasti. Lisäksi käyttötapauksia voidaan muuttaa, hylätä tai lisätä projektin aikana. Projektin alustavia kustannus- ja aikatauluarvioita voidaan päivittää projektin edetessä. Työmäärä ja kustannus voidaan

arvioida paremmin yksittäistä lyhyttä iteraatiota varten. Mahdollisuus vaatimusten, projektin laajuuden ja prioriteetin muutoksille tekee tarkkojen kustannus- ja työmääräarvioiden antamisesta koko projektille haasteellista (Cao ja Ramesh, 2008; Ramesh et al., 2010; Savolainen et al., 2010).

Haaste	**Kuvaus**
Ongelmat liittyen asiakkaaseen tai asiakasedustajiin	Asiakaskommunikaatiossa, luottamuksessa tai yhteydessä oikeisiin ja hyviin asiakasedustajiin voi esiintyä ongelmia.
Käyttäjätarinoiden riittämättömyys	Käyttäjätarinoilla ei voida kattavasti kuvata kaikkea järjestelmien rakentamiseen vaadittavaa informaatiota.
Haasteet vaatimusten priorisoinnissa	Vaatimusten välitön arvo tärkeimpänä prioriteettina ei välttämättä ota huomioon tärkeitä, mutta näkymättömämpiä vaatimuksia. Asiakkaan kyky priorisoida vaatimuksia voi olla haasteellinen.
Kasvava tekninen velka	Lyhyen tähtäimen suunnitelmat voivat johtaa teknisiin ratkaisuihin, jotka eivät pysty vastaamaan enää uusien tai myöhemmin toteutettavien vaatimusten tarpeisiin.
Tukeutuminen hiljaiseen tietoon vaatimuksista	Hiljaisen tiedon määrä on suuri, kun taas tuotettavan dokumentaation määrä on minimaalinen. Henkilöstömuutokset voivat aiheuttaa tiedon katoamista.
Epätarkat työmääräarviot	Aikataulujen ja kustannusten arvioiminen on haasteellista ilman kattavaa vaatimusten spesifiointia ja analysointia.

Taulukko 4.6: Yhteenveto ketterän vaatimusmäärittelyn esitetyistä haasteista (Heikkilä et al., 2015)

4.3.3 Vaatimusmäärittelyn käytänteitä

Tässä kappaleessa käydään läpi, minkälaisilla ketterän vaatimusmäärittelyn käytänteillä voidaan tehdä vaatimusmäärittelyä ketterissä ohjelmistoprojekteissa. Näillä käytänteillä voidaan suorittaa perinteisen vaatimusmäärittelyprosessin kaltaisia vaiheita eli ne tukevat vaatimusten määrittelyä ketteriä menetelmiä käytettäessä. Ketterän vaatimusmäärittelyn tavat määritellä vaatimuksia eivät vastaa tai seuraa perinteisen vaatimusmäärittelyyn liittyviä oletuksia tai prosesseja. Vaatimusmäärittelyyn liittyviä käytänteitä on erikseen

tunnistettu tutkimuskirjallisuudessa, sillä varsinaiset ketterät menetelmät eivät ota vaatimusmäärittelyyn suoraan kantaa (Heikkilä et al., 2015). Ketterään vaatimusmäärittelyyn liittyviä käytänteitä ja niiden vaikutuksia on erikseen tunnistettu ja tutkittu (Cao ja Ramesh, 2008; Lucia ja Qusef, 2010; Ramesh et al., 2010; Paetsch et al., 2003; Inayat et al., 2015). Tässä alaluvussa käydään läpi Ramesh et al. (2010) tutkimuksessaan tunnistamat kuusi ketterän vaatimusmäärittelyn käytännettä, joita käytettiin 16:ssa ketteriä menetelmiä hyödyntävässä yrityksessä. Tutkimuksessaan he myös arvioivat käytänteiden suhdetta perinteiseen vaatimusmäärittelyprosessiin, selvittivät ketterään vaatimusmäärittelyyn liittyviä haasteita ja arvioivat käytänteiden ja haasteiden vaikutuksia vaatimuksiin liittyviin ohjelmistoprojektin riskeihin. Tästä syystä kyseinen lähestymistapa on myös valittu tähän tutkielmaan.

Ramesh et al. (2010) raportoimat ketterän vaatimusmäärittelyn käytänteet ovat:

- kasvokkain käytävä kommunikaatio kirjallisen spesifikaation sijaan
- iteratiivinen vaatimusmäärittely
- jatkuva vaatimusten priorisointi
- vaatimusten muuttumisen hallinta jatkuvan suunnittelun avulla
- prototyypitys
- katselmointipalaverit ja hyväksymistestaus.

Kasvokkain käytävä kommunikaatio kirjallisen spesifikaation sijaan

Ketterät menetelmät suosivat kasvokkain käytävää kommunikaatiota kirjoitetun dokumentaation sijaan (Agile Alliance, 2001). Kasvokkain tapahtuva kommunikaation avulla kehitystiimi saa tietoa asiakkaalta ilman kattavan vaatimusdokumentaation luomista. Asiakkaalla on mahdollisuus ohjata projektia erilaisiin suuntiin. Korkealla tasolla olevia vaatimuksia voidaan kuvata käyttäjätarinoilla. Kuvaukset ovat lähtökohtana yksityiskohtaisemmalle keskustelulle asiakkaan kanssa ennen kehitystyön aloittamista ja sen aikana. Kommunikaatio vähentää tarvetta tehdä aikaa vieviä dokumentaatio- ja hyväksymisprosesseja. Olemassa oleva vaatimusdokumentaatio ei poista tarvetta keskustelulle, sillä vaatimukset voivat silti olla esimerkiksi moniselitteisiä (Cao ja Ramesh, 2008; Ramesh et al., 2010).

Iteratiivinen vaatimusmäärittely

Iteratiivista vaatimusmäärittelyä tehdään toistuvasti koko ohjelmistokehitysprosessin ajan jokaisessa kehitystyön syklissä. Asiakas tapaa kehitystiimin antaakseen tarkkaa ja yksityiskohtaisempaa tietoa seuraavaksi toteutettavista ominaisuuksista (Cao ja Ramesh, 2008).

Ketterissä ohjelmistoprojekteissa kaikki projektin vaatimukset eivät ole ennalta määriteltyjä projektin alkaessa. Projektin alussa selvitetään karkeasti järjestelmän kriittisimmät vaatimukset, jotka toimivat lähtökohtana kehitykselle ja ensimmäisen kehityssyklin suunnittelulle. Projektin edetessä vaatimukset tarkentuvat ja niitä lisätään tarpeen mukaan (Ramesh et al., 2010). Iteratiivinen vaatimusmäärittely voi johtaa tyydyttävämpään asiakassuhteen sekä selkeämpiin ja paremmin ymmärrettäviin vaatimuksiin (Cao ja Ramesh, 2008).

Asiakkaat eivät välttämättä tiedosta tai osaa artikuloida kaikkia vaatimuksia projektin alussa. Vaatimukset muuttuvat ja kehittyvät esimerkiksi ympäristön muutoksen, asiakkaan kasvavan käsityksen tai ennalta arvaamattomien teknisten ongelmien vuoksi. Joustava vaatimusmäärittely mahdollistaa erilaisten ratkaisujen löytämisen (Cao ja Ramesh, 2008). Syitä kehitystyön aloittamiselle ilman aikaa vievää vaatimusten analysointia voi olla monia (Ramesh et al., 2010). Esimerkiksi vaatimusten epävakaus, teknisten yksityiskohtien epäselvyys ja asiakkaan kykenemättömyys ilmaista vaatimuksia näkemättä niitä voivat olla perusteluita sille, ettei vaatimuksia määritellä kattavasti projektin alussa.

Jatkuva vaatimusten priorisointi

Jatkuvaa vaatimusten priorisointia tehdään koko projektin ajan, jotta asiakkaalle voidaan toteuttaa eniten arvoa tuottavia ominaisuuksia mahdollisimman nopeasti jokaisessa kehityssyklissä. Perinteiset menetelmät priorisoivat tyypillisesti vaatimukset ainoastaan kerran vaatimusten analysointivaiheessa. Ketterät menetelmät priorisoivat vaatimuksia aina tarvittaessa uudelleen. Vaatimukset priorisoidaan muiden kehitystehtävien, kuten esimerkiksi korjaus- ja muutostöiden sekä refaktoroinnin kanssa (Ramesh et al., 2010). Vaatimusten priorisointi on oleellista analysoitaessa vaatimuksia projekteille, joilla rajalliset resurssit budjetin, työntekijöiden ja aikataulun suhteen (Sommerville ja Sawyer, 1997).

Perinteisissä vaatimusmäärittelyn menetelmissä vaatimusten prioriteettiin vaikuttavat monet eri tekijät kuten liiketoiminnallinen arvo, riskit, hinta ja toteutuksen riippuvuudet (Kotonya ja Sommerville, 1998). Ketterissä menetelmissä priorisoinnissa käytetään pääasiassa asiakkaan vaatimuksille määrittelemää arvoa (Ramesh et al., 2010).

Asiakkaiden läsnäolo kehitysprosessissa auttaa kehitystiimiä saamaan tarkan ymmärryksen asiakkaan liiketoiminnallisista prioriteeteista, mikä taas auttaa kehitystiimiä täyttämään asiakkaan tarpeet paremmin. Ketterät vaatimusmäärittelymenetelmät tarjoavat – perinteisistä menetelmistä poiketen – useita mahdollisuuksia vaatimuksien uudelleenpriorisoimiselle (Cao ja Ramesh, 2008).

Vaatimusten muuttumisen hallinta jatkuvan suunnittelun avulla

Muutoksien sallimisella vaatimuksiin kehitystyön aikana pyritään saavuttamaan paremmin asiakkaan tarpeita vastaava järjestelmä. Muutokset ovat helpommin toteutettavissa ja halvempia ketteriä menetelmiä käytettäessä (Cao ja Ramesh, 2008).

Ominaisuuksien lisääminen tai poistaminen sekä muutokset jo toteutettuihin ominaisuuksiin ovat tyypillisiä ketterässä ohjelmistokehityksessä tapahtuvia muutoksia vaatimuksiin. Jokaisen kehityssyklin päätteeksi asiakkaat katselmoivat valmiit ominaisuudet ja voivat pyytää niihin muutoksia, jos heidän odotuksensa eivät täyty. Asiakkaan ja kehittäjien kommunikoivat paljon ennen implementaatiota ja sen aikana, minkä johdosta muutokset valmiisiin ominaisuuksiin tässä kohtaa ovat kuitenkin harvinaisia (Ramesh et al., 2010). Kommunikaation puuttuessa suuretkin muutokset voivat kuitenkin olla mahdollisia.

Aikaisen ja jatkuvan validoinnin ja tiiviin kommunikaation seurauksena tarve suurille muutoksille pienenee merkittävästi. Tästä johtuen muutospyynnön kustannus on pienempi ketteriä menetelmiä käytettäessä kuin perinteisiä menetelmiä käytettäessä (Cao ja Ramesh, 2008).

Prototyypitys

Prototyyppien kehittäminen voi nopeasti selkeyttää vaatimusten analysointia asiakkaalta saatavan palautteen perusteella. Dokumentaation sijaan voidaan käyttää prototyyppiä vaatimusten validointiin ja parantelemiseen asiakkaan kanssa. Tuotanto-ohjelmistokin voi tiettyyn pisteeseen asti toimia eräänlaisena prototyyppinä. Kiire markkinoille sekä mahdollisuus julkaista nopeasti uusia versioita suosivat prototyyppien käyttämistä suoraan tuotannossa vakaamman ja kestävämmän toteutuksen sijaan (Cao ja Ramesh, 2008; Ramesh et al., 2010). Prototyyppien käyttämisessä tuotannossa voi kuitenkin seurata riskejä esimerkiksi skaalautuvuuteen, tietoturvaan ja jatkokehitykseen liittyen.

Katselmointipalaverit ja hyväksymistestaus

Vaatimusten validoimiseksi voidaan käyttää kehityssyklin lopuksi pidettäviä katselmointipalavereja ja vaatimusten täyttymistä testaavia hyväksymistestejä. Katselmoinneissa projektin tilanne raportoidaan asiakkaalle ja muille sidosryhmille. Katselmoinnin aikana kehittäjät esittelevät syklin aikana toimitettuja ominaisuuksia. Asiakas ja laadunvarmistuksesta vastaavat henkilöt esittävät kysymyksiä ja antava palautetta (Cao ja Ramesh, 2008). Katselmointipalavereilla on muun muassa mahdollisuus varmistaa projektin edistyvän aikataulun ja halutun suunnitelman mukaisesti, lisätä asiakkaan luottamusta kehitystiimiä ja projektia kohtaan sekä huomata mahdolliset ongelmat jo varhaisessa kehitysvaiheessa (Ramesh et al., 2010). Katselmoinneissa huomataan harvoin suuria ongelmia (Ramesh et al., 2010), sillä vaatimusten validointi tapahtuu suurelta osin asiakaskommunikaation kautta ennen kehityssyklin alkua ja sen aikana.

Hyväksymistestit, jotka asiakas toimittaa tai kehittää laadunvarmistuksesta vastaavan henkilön kanssa, ovat toinen tapa validoida vaatimuksia. Hyväksymistestit voidaan nähdä myös osana vaatimusspesifikaatiota (Cao ja Ramesh, 2008). Myös testivetoista ohjelmistokehitystä (TDD) voidaan käyttää vaatimusten validointiin. Sen lähtökohtana on testitapauksen luominen ennen varsinaisen toiminnallisuuden toteuttavaa ohjelmakoodia (Beck, 2000). Kirjoitettavat testit ovat osa vaatimusmäärittely- ja suunnitteluprosessia, sillä niiden avulla ensin määritellään tarkasti ohjelmiston haluttu toiminta. Muutoksien tekeminen helpottuu, sillä testit kertovat välittömästi täyttyvätkö ohjelmiston vaatimukset vielä muutoksen jälkeenkin (Cao ja Ramesh, 2008).

4.3.4 Vaikutukset vaatimusten riskeihin

Ohjelmistoprojekteihin liittyy useita erityyppisiä riskejä, jotka voivat vaikuttaa projektin aikatauluihin tai toimitettavan ohjelmiston laatuun (Sommerville, 2007). Ketterillä vaatimusmäärittelyn käytänteillä voidaan vaikuttaa ohjelmistoprojektin vaatimuksiin liittyviin riskeihin, mutta tietyissä tilanteissa riskit saattavat myös kasvaa. Kartoittamalla projektin vaatimuksiin liittyviä riskejä voidaan muodostaa kuva ketterän tai perinteisen menetelmän soveltuvuudesta projektiin (Ramesh et al., 2010). Riskit voidaan karkeasti jakaa projektin, tuotteen ja liiketoiminnan riskeihin (Sommerville, 2007). Projektin riskit vaikuttavat sen aikatauluun tai resursseihin. Tuotteen riskit vaikuttavat tuotettavan ohjelmiston laatuun tai tehokkuuteen. Liiketoiminnan riskit vaikuttavat ohjelmistoa tuottavaan tai hankkivaan tahoon. Yksittäiset riskit voivat kuulua yhteen tai useampaan näistä kategorioista.

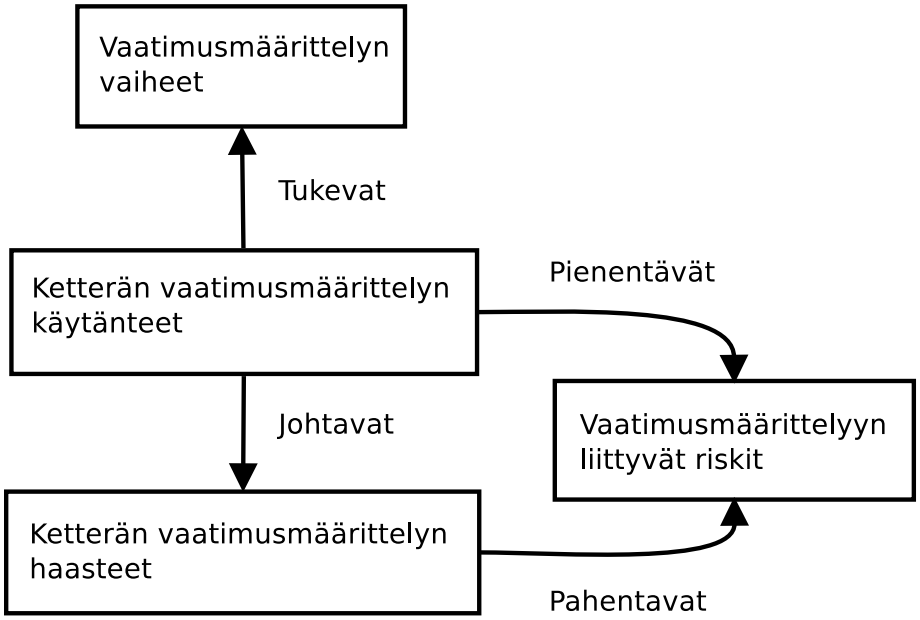
Riskienhallinnan avulla riskien vaikutukset projektiin voidaan pyrkiä minimoimaan ja projektin riskit voidaan tunnistaa ja analysoida sekä niihin voidaan varautua ja niitä voidaan tarkkailla.

Ramesh et al. (2010) tunnistivat tutkimuksessaan 12 muussa kirjallisuudessa mainittua vaatimuksiin liittyvää ohjelmistoprojektin riskiä, joista he koostivat yhdeksän erillistä riskiä. He analysoivat tunnistamiensa ketterän vaatimusmäärittelyn käytänteiden ja niihin liittyvien vaatimusmäärittelyn haasteiden vaikutukset näihin riskeihin. Ramesh et al. (2010) tunnistamat ketterän vaatimusmäärittelyn haasteet olivat huomioituna myös Heikkilä et al. (2015) tutkimuksessa (katso luku 4.3.2).

Tunnistetut ohjelmistoprojektin vaatimuksiin liittyvät riskit ovat (Ramesh et al., 2010):

- Puuttuvat vaatimukset ja vaatimusten epävakaus
- Ongelmat sidosryhmien pätevyyden ja yksimielisyyden kanssa — Sidosryhmät eivät saa yhteistä käsitystä vaatimuksista ja asiakas ei ole yhteistyöhaluinen, edustava, auktorisoitu tekemään päätöksiä, sitoutunut tai perillä asioista.
- Riittämätön asiakkaan ja kehittäjien vuorovaikutus
- Tärkeän vaatimuksen vähäinen huomioiminen
- Pelkkien toiminnallisten vaatimuksien mallintaminen
- Vaatimuksien jättäminen tarkastamatta — Vaatimuksen vaikutuksia muihin vaatimuksiin ei huomioida riittävästi.
- Tarkan vaatimuksen esittäminen toteutuksen muodossa — Vaatimuksia ei kuvata tarkasti kirjallisesti, joten toteutukset kuvaavat lopullista vaatimusta.
- Pyrkimys saada vaatimukset kattaviksi ennen toteutusta
- Aikataulutusvirheet — Ongelmat kustannus- ja työmääräarvioissa tekevät aikataulun arvioinnista haastavaa.

Ramesh et al. (2010) tutkivat tunnistamiensa ketterän vaatimusmäärittelyn käytänteiden vaikutusta ohjelmistoprojektin vaatimuksiin liittyviin riskeihin. He huomasivat, että jotkin käytänteistä pienensivät näitä riskejä, mutta jotkin näistä käytänteistä seuranneista haasteista pahensivat vaatimuksiin liittyviä riskejä. Kuvassa 4.3 on esitetty nämä suhteet selvemmin.



Kuva 4.3: Malli ketterän vaatimusmäärittelyn käytänteiden vaikutuksista (Ramesh et al., 2010)

Ketterän vaatimusmäärittelyn käytänteillä voidaan pienentää projektin vaatimuksiin liittyviä riskejä. Joihinkin riskeihin niillä on kuitenkin pahentava tai vaihteleva vaikutus (Ramesh et al., 2010). Ketterän vaatimusmäärittelyn käytänteillä on mahdollista pienentää kolmea tunnistettua riskiä: vaatimusten puuttumista, tärkeiden vaatimusten heikkoa huomioimista ja pyrkimystä tehdä vaatimuksista kattavia ennen toteutusta. Kolmea riskiä käytänteet kasvattavat: toiminnallisten vaatimusten ylikorostumista, riittämätöntä vaatimusten tarkastusta sekä vaatimusten esittämistä toteutuksen muodossa. Käytänteillä on vaihteleva vaikutus jäljellä olevaan kolmeen riskiin: sidosryhmien pätevyyteen ja yksimielisyyteen, riittämättömään asiakkaan ja kehittäjien vuorovaikutukseen sekä aikataulutusvirheisiin.

Ramesh et al. (2010) jakoivat vaatimusmäärittelyn riskit myös mukautuviin ja hankaliin riskeihin sen perusteella, miten ketterän vaatimusmäärittelyn käytänteet ja haasteet yhdessä joko pienentävät tai suurentavat niiden todennäköisyyttä. Mukautuviin riskeihin pystytään vastaamaan helposti. Hankalista riskeistä on puolestaan vaikeampi selvitä ketterien vaatimusmäärittelyn käytänteiden avulla. Kaksi tunnistettua hankalaa riskiä olivat ongelmat sidosryhmien pätevyyden ja yksimielisyyden kanssa sekä pelkkien toiminnallisten vaatimuksien mallintaminen.

Ramesh et al. (2010) tutkimuksen perusteella voidaan todeta, että ketterän vaatimusmäärittelyn käytänteiden vaikutukset riskeihin kannattaisi suorittaa projektikohtaisesti vähin-

tään projektin alkaessa, jotta pystytään tunnistamaan tarve esimerkiksi perinteisemmälle vaatimusmäärittelyprosessille. Ketterällä vaatimusmäärittelyllä ei siis voida ratkaista kaikkia ohjelmistoprojektin vaatimusmäärittelyyn liittyviä ongelmia.

Taulukko 4.7 esittää vielä yksityiskohtaisemmin Ramesh et al. (2010) tapaustutkimuksen tulokset ketterän vaatimusmäärittelyn käytänteiden ja haasteiden suhteesta ohjelmistoprojektin vaatimuksiin liittyviin riskeihin. Taulukossa ongelman luonne kuvaa Ramesh et al. (2010) määrittämää mukautuvuutta. Mukautuvuus riippuu siitä, liittyykö riski ketterän vaatimusmäärittelyn käytänteeseen vai haasteeseen ja oliko käytänne tai haaste tapaustutkimuksen aineiston perusteella oleellinen.

Vaatimusmäärittelyn riski	**Liittyvä ketterän vaatimusmäärittelyn käytänne tai haaste**	**Käytänteen tai haasteen vaikutus riskeihin**	**Vaikutuksen suuruus**	**Ongelman luonne**
Puuttuvat vaatimukset ja vaatimusten epävakaus	+ Kasvokkain käytävä kommunikaatio + Iteratiivinen vaatimusmäärittely + Jatkuva suunnittelu	Pienentää	Keskisuuri–suuri	Mukautuva
Ongelmat sidosryhmien pätevyyden ja yksimielisyyden kanssa	+ Iteratiivinen vaatimusmäärittely – Pääsy asiakkaaseen ja asiakkaan osallistuminen	Vaihteleva	Suuri	Hankala
Riittämätön asiakkaan ja kehittäjien vuorovaikutus	+ Iteratiivinen vaatimusmäärittely – Pääsy asiakkaaseen ja asiakkaan osallistuminen	Vaihteleva	Suuri	Mukautuva
Tärkeän vaatimuksen vähäinen huomioiminen	+ Vaatimusten priorisointi + Arviointipalaverit ja testit	Pienentää	Keskisuuri–suuri	Mukautuva
Pelkkien toiminnallisten vaatimuksien mallintaminen	– Ei-toiminnallisten vaatimuksien laiminlyönti	Vaikeuttaa	Matala	Hankala
Vaatimuksien jättäminen tarkastamatta	– Ei verifiointia	Vaikeuttaa	Suuri/matala	Mukautuva
Tarkan vaatimuksen esittäminen toteutuksen muodossa	– Minimaalinen dokumentaatio	Vaikeuttaa	Keskisuuri–suuri	Mukautuva
Pyrkimys saada vaatimukset kattaviksi ennen toteutusta	+ Iteratiivinen vaatimusmäärittely	Pienentää	Keskisuuri	Mukautuva
Aikataulutusvirheet	+ Jatkuva suunnittelu – Ongelmat hinnan ja aikataulujen arvioissa	Vaihteleva	Matala–keskisuuri	Mukautuva

Taulukko 4.7: Tunnistettujen ketterän vaatimusmäärittelyn käytänteiden vaikutukset ohjelmistovaatimuksiin liittyviin projektin riskeihin (Ramesh et al., 2010)
+: vaatimusmäärittelyn käytänne
–: vaatimusmäärittelyn haaste

5 Tapaustutkimus: Software Factory – stonehenge2

Tämä tutkielma kokonaisuudessaan pyrkii tarjoamaan näkemyksen siihen, miten ketterä vaatimusmäärittely vaikuttaa vaatimusten muodostumiseen ohjelmistokehityksessä. Tässä luvussa kuvataan tarkemmin tutkimusasetelma, tutkimus- sekä tiedonkeruumenetelmät. Tapaustutkimuksen analyysin tulokset esitetään luvussa 6.

Tässä tapaustutkimuksessa selvitetään, voidaanko tutkimuskirjallisuudessa mainittuja ketterän vaatimusmäärittelyyn hyötyjä sekä haasteita (katso luku 4.3.2) havaita ketterillä menetelmillä toteutetussa ohjelmistoprojektissa. Tapaustutkimuksen avulla voidaan saada ymmärrystä siitä, miten ilmiö, kuten vaatimusmäärittely, näyttäytyy ketterän ohjelmistoprojektin kaltaisessa ympäristössä (Runeson ja Höst, 2009; Hirsjärvi et al., 2000). Sen kautta voidaan syventää ymmärrystä tutkittavasta ilmiöstä ja saada myös yksityiskohtaista, intensiivistä tietoa siitä.

Tämä tutkielma noudattaa ohjelmistotuotantoprojekteista raportoiville tapaustutkimuksille suositeltua lineaaris-analyyttistä lähestymistapaa, jossa tutkielman rakenne jäsennetään seuraavasti: lähtökohdat tutkimukselle, aikaisempi tutkimus, tutkimusmenetelmät, tulokset ja johtopäätökset (Runeson ja Höst, 2009). Tapaustutkimus on tehty noudattaen Per Runesonin ja Martin Höstin artikkelissaan *Guidelines for conducting and reporting case study research in software engineering* antamaa ohjeistusta (Runeson ja Höst, 2009).

Tapaustutkimus koostuu ohjelmistokehitysprojektista, joka suoritettiin Helsingin yliopistolla sijaitsevassa Software Factoryssä. Tämän tutkielman empiirisen osan tarkoitus on testata tutkimuskirjallisuudessa esitettyjä tuloksia ketterän vaatimusmäärittelyn hyödyistä ja haasteista varmentaen, voidaanko ohjelmistoprojektissa tehdä samanlaisia havaintoja. Oletuksena on, että samanlaisia hyötyjä ja haasteita kyetään tunnistamaan ja havaitsemaan myös tapaustutkimuksen projektissa.

5.1 Tutkimusympäristö

Tämä tapaustutkimus tehtiin vuoden 2012 lopussa järjestetyllä Software Factory Project -kurssilla (*Software Factory – University of Helsinki* 2018). Helsingin yliopiston tietojenkäsittelytieteen laitoksella toimiva Software Factory* on perustettu tammikuussa 2010. Sellaisia löytyy Helsingin lisäksi myös muualta Suomesta sekä Euroopasta. Se on kokeellinen ohjelmistotuotannon tutkimuslaboratorio, jossa asiakkaat, opiskelijat ja tutkijat toimivat yhdessä. Software Factory on fyysinen tila, jonka työolosuhteet vastaavat hyvin paljon oikeaa työympäristöä. Yksi projekti kestää tyypillisesti seitsemän viikkoa.

Opiskelijoille kurssi on tilaisuus päästä työskentelemään ohjelmistoprojektiin, jossa tuotetaan asiakkaalle ohjelmisto perusteltuun tarpeeseen. Se on myös mahdollisuus työskennellä intensiivisesti testaten ja oppien ohjelmistokehityksessä vaadittavia taitoja. Asiakkaat voivat projektin myötä esimerkiksi saada tuotettua prototyypin ideastaan tai saada uusia näkemyksiä ohjelmistokehityksestä. Software Factory on myös ympäristö, jossa voidaan siellä tehtävien projektien kautta tehdä empiiristä tutkimusta ohjelmistotuotannon ja muidenkin tieteenalojen näkökulmista monilla eri tutkimusmenetelmillä.

Opiskelijat saavat kurssille osallistumisesta opintopisteitä työskentelypäivien mukaisesti. Neljäpäiväistä työviikkoa tekevät saavat 10 opintopistettä ja viisipäiväistä tekevät 12 opintopistettä. Jokainen työpäivä on noin kuuden tunnin mittainen. Rahallista hyötyä kurssille osallistumisesta ei saa.

Software Factoryssä suositaan yleensä projektista riippumatta ketteriä prosessimalleja. Tämä tarkoittaa muun muassa Scrum- ja Kanban-menetelmien hyödyntämistä projekteissa.

5.2 Tiedonkeruumenetelmät

Tapaustutkimuksen kohteena olevan projektin aikana kerätty tutkimusaineisto koostuu kokonaisuudessaan osallistuvan havainnoinnin aikana kirjoittajan tekemistä muistiinpanoista, Kanban-taulun läpikäyneistä tehtävistä (post-it-lapuista), Git-versionhallinnan sisällöstä sekä kurssin päätteeksi järjestetyn lopputapaamisen aikana asiakkaan kanssa luoduista projektia kuvaavista kuvista.

Kyselyiden ja haastattelun avulla voidaan selvittää, miten tutkittavat kohteet havaitsevat

*https://www.softwarefactory.cc/

tapahtumat ympärillään. Sillä ei saada välttämättä selville, mitä todella tapahtuu. Havainnoinnin avulla puolestaan voidaan selvittää, toimivatko ihmiset todella niin kuin he väittävät sekä mitä tutkittavassa ympäristössä tapahtuu (Hirsjärvi et al., 2000).

Havainnoinnin avulla voidaan saavuttaa syvällinen ymmärrys tutkittavasta ilmiöstä (Runeson ja Höst, 2009). Sen avulla voidaan kerätä monipuolista, välitöntä ja suoraa tietoa tutkittavan kohteen toiminnasta ja käyttäytymisestä. Havainnoinnilla saadaan myös tietoa tilanteista, jotka ovat vaikeasti ennakoitavissa tai ovat nopeasti muuttuvia. Se sopii myös hyvin ihmisten vuorovaikutuksen tutkimiseen (Hirsjärvi et al., 2000).

Osallistuvassa havainnoinnissa tutkija ei ole pelkkä passiivinen tarkkailija, vaan pyrkii osaksi tarkkailtavaa ryhmää (Hirsjärvi et al., 2000). Sen avulla voidaan tutkia, miten ohjelmistokehittäjät toimivat tietyissä tilanteissa (Runeson ja Höst, 2009).

Riskinä osallistuvassa havainnoinnissa on, että tietoa voi olla joissakin tilanteissa vaikeaa tallentaa välittömästi, jolloin tutkija joutuu luottamaan muistiinsa kirjatessaan havaintoa myöhemmin. Osallistujan rooli voi siis vaatia liikaa huomiota havainnoitsijan rooliin verrattuna (Hirsjärvi et al., 2000).

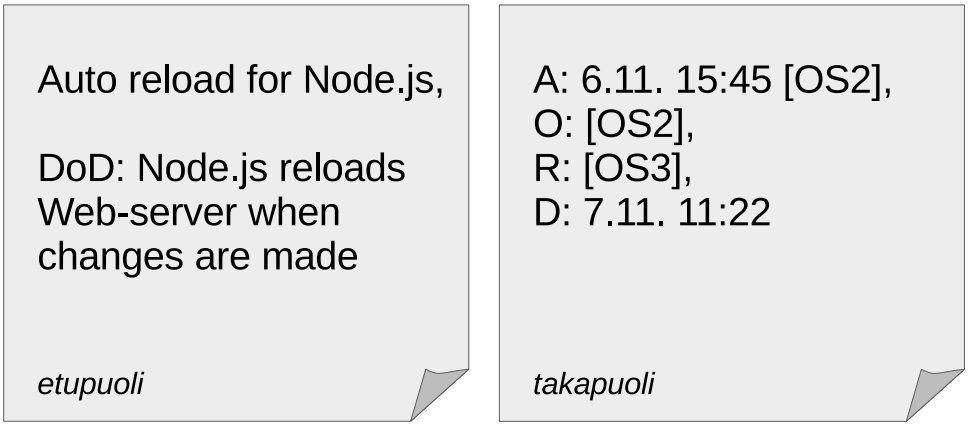
Kirjoittaja teki osallistuvaa havainnointia osallistumalla tapaustutkimuksen projektiin itse opiskelijana osana kehitystiimiä, joka kehitti asiakkaalle ohjelmistoa. Kirjoittaja teki päivittäisiä muutaman lauseen mittaisia muistiinpanoja projektin aikana. Muistiinpanoihin kirjattiin päivän aikana tapahtuneet tapahtumat sekä erityisesti vaatimusmäärittelyn kannalta oleelliset huomiot. Myös yleisiä ajatuksia siitä, mitä tapahtui, miten ihmiset käyttäytyivät ja miten projekti eteni, kirjattiin muistiinpanoihin. Kirjoittaja osallistui myös samalle asiakkaalle tehdyn aiemman projektin lopputapaamiseen kirjaten muistiinpanoja tapaamisen pääsisällöistä. Kirjoittajalla ei vielä projektin alkaessa ollut tätä tutkielmaa vastaavaa tietomäärää ketterän vaatimusmäärittelyn erityispiirteistä.

Osallistuvaan havainnointiin liittyviä riskejä voi pienentää käyttämällä useita tietolähteitä (Runeson ja Höst, 2009). Useiden tietolähteiden käyttäminen on tärkeää empiirisessä kvalitatiivisessa tutkimuksessa, jotta saadaan uusia näkökulmia ja jotta sama ilmiö on mahdollista varmistaa tarvittaessa myös toisesta tietolähteestä. Johtopäätös on merkittävämpi, jos se voidaan johtaa useammasta tietolähteestä. Muistiinpanojen lisäksi tässä tapaustutkimuksessa käytettiin muitakin tietolähteitä.

Software Factoryn projektissa yksittäiset työtehtävät kirjattiin post-it-lapuille, jotka kulkivat fyysisen Kanban-taulun läpi määritellyn prosessimallin mukaisesti. Ne kerättiin manuaalisesti talteen tutkimustarkoituksia varten. Projektin alkaessa kehitystiimin jäseniä oh-

jeistettiin merkitsemään jokaiseen työtehtävään tietoja tutkimustarkoituksia varten. Työtehtävän takapuolella löytyvät tiedot tilasiirtymistä Kanban-taululla. Opiskelijat kirjasivat tehtävän aloittamisen ajankohdan, joka tarkoitti tilasiirtymää *TODO*-sarakkeesta *Analyze*-sarakkeeseen. Lappuun kirjattiin myös tehtävän valmistumisen ajankohta, joka tarkoitti tilasiirtymää *Review*-sarakkeesta *Done*-sarakkeeseen. Jokainen opiskelija kirjasi myös nimimerkkinsä niihin työvaiheisiin, joiden tekemiseen hän oli osallistunut eli joko *Analyze*, *Ongoing* tai *Review*. Kanban-taulun sarakkeiden rakenne ja järjestys oli projektin keston ajan: *TODO*, *Analyze*, *Ongoing*, *Review*, *Done*. Esimerkki Kanban-taulun läpi kulkeneesta työtehtävästä löytyy kuvasta 5.1 ja 6.2.

Tikettien takapuolelle kirjattuja tilasiirtymien aikaleimoja tai tietyn vaiheen suorittaneiden henkilöiden tietoja ei hyödynnetty projektin aikana esimerkiksi tikettien läpimenoaikojen seuraamiseen. Tiedot kirjattiin ainoastaan myöhempiä tutkimustarkoituksia varten. Hylättyjä työtehtäviä eli niitä, jotka eivät päätyneet *done*-sarakkeeseen, ei kerätty osaksi aineistoa eikä niitä projektin aikana säästetty muitakaan tarkoituksia varten. Työtehtävät hylättiin usein siirrettäessä tehtäviä *analyze*-sarakkeeseen. Syitä hylkäämiselle olivat esimerkiksi tehtävän toteaminen turhaksi tai sen yhdistyminen toisen tehtävän kanssa.



Kuva 5.1: Esimerkki Kanban-taulun läpi kulkeneesta vaatimuksesta liittyen suorituskykytestaussovelluksen mittaustuloksia näyttävään osaan. Tekijöiden nimet on koodattu *OS2* ja *OS3*. Takapuolelle kirjatut tilasiirtymät ovat: **A**nalyze, **O**ngoing, **R**eview ja **D**one.

Projektin aikana tuotettu lähdekoodi tallennettiin Git-versionhallintajärjestelmään*. Tämän tietolähteen avulla Kanban-taulun läpi virranneiden tehtävien yhteys koodiin oli tarvittaessa mahdollista tunnistaa manuaalisesti tiketteihin kirjattujen aikaleimojen ja tekijätietojen perusteella.

*Git https://git-scm.com/

Projektin lopuksi pidettiin asiakkaan ja projektiin osallistuneiden opiskelijoiden kesken lopputapaaminen, jossa käytiin projektin vaiheita läpi sekä reflektoitiin projektin tapahtumia. Tapaaminen pidettiin projektin valmistumisen jälkeen, kun ohjelmisto oli valmis ja toimitettu asiakkaalle. Tapaamisen aikana asiakkaan edustajat ja projektin kehitystiimi piirsivät yhdessä kolme eri kuvaa projektista, jotka kuvasivat asiakkaan ja kehittäjien yhteisiä näkemyksiä. Ensimmäisessä kuva esitti projektin aikajanaa, jolle merkittiin projektin tärkeimmät tapahtumat. Toisessa kuvassa listattiin projektin ohjelmistossa olevat eri teknologiat ja niiden suhteet toisiinsa. Viimeisessä kuvassa esitettiin, millaisia henkilökohtaisia motivaattoreita projektissa koettiin. Tapaamisessa muodostettu aikajana oli hyvä lähtökohta varmistamaan muistiinpanoista sellaisia tapahtumia tai käännekohtia, jotka asiakkaan edustajat tai kehitystiimin jäsenet kokivat tärkeiksi projektissa sekä sellaisia, jotka olivat saattaneet jäädä kirjaamatta muistiinpanoihin.

Tutkimuksen raportoiduista tuloksista on yksityisyyssyistä anonymisoitu suorat tunnistettavat nimet asiakkaan, opiskelijoiden ja muiden projektiin osallistuneiden osalta, jotta yksittäisten henkilöiden tunnistaminen ei ole tuloksista mahdollista (Runeson ja Höst, 2009).

6 Havaintotulokset

Tässä pääluvussa käydään läpi tapaustutkimuksen tulokset. Aluksi käydään läpi projektin perustietoja tarkastelemalla luvussa 6.1, miten työ eteni projektissa sekä minkälainen työskentelyprosessi siinä oli. Luvussa selvennetään myös, miten vaatimukset syntyivät projektissa ja miten niitä hallittiin. Seuraavaksi luvussa 6.2 kerrotaan, miten ketterään vaatimusmäärittelyyn liitetyt hyödyt ja haasteet olivat havaittavissa tapaustutkimuksen kohteena olevassa ohjelmistoprojektissa. Tämä tehdään tarkastelemalla, miten tutkimuskirjallisuudessa (katso luku 4.3.2) esitetyt hyödyt sekä haasteet ilmenivät projektissa. Lopuksi esitetään vielä yhteenveto analyysin tuloksista.

6.1 Projektin kuvaus

Tässä alaluvussa käydään läpi projektin perustietoja sekä miten työ projektissa suoritettiin. Tämä on oleellista ja taustoittavaa tietoa luvun 6.2 analyysin tuloksille.

6.1.1 Projektin perustiedot

Tapaustutkimuksen kohteena on Helsingin yliopiston tietojenkäsittelytieteen laitoksen toisen periodin aikana 29.10.–14.12.2012 järjestetty Software Factory Project -kurssi (katso luku 5.1), jonka aikana toteutettiin asiakasprojekti sulautettua internet-videopuhelujärjestelmää kehittävälle startup-yritykselle. Projekti kesti yhteensä 35 työpäivää, joista ryhmä piti samanaikaisesti vapaata itsenäisyyspäivän 6.12. sekä kaksi muuta päivää. Projektissa toteutettiin asiakkaalle hänen tarpeitaan vastaava ohjelmisto. Ryhmä työskenteli koko projektin ajan Software Factoryn tiloissa projektille varatussa huoneessa. Tämän tapaustutkimuksen kohteena on projekti *stonehenge2*. Samalle asiakkaalle oli tehty projekti samalla kurssilla myös edellisessä opetusperiodissa.

Kurssille osallistui yhteensä viisi opiskelijaa. Yksi opiskelijoista päätti jättää kurssin kesken projektin toisella viikolla. Jäljelle jääneet neljä opiskelijaa työskentelivät projektissa täyspäiväisesti sen päättymiseen saakka. Kirjoittaja itse osallistui tapaustutkimuksen projektiin opiskelijana osana kehitystiimiä, joka kehitti asiakkaalle ohjelmistoa. Kurssille osallistui kyseisellä järjestämiskerralla opinnoissaan jo loppusuoralla olevia opiskelijoita.

Osallistujille oli kertynyt jo jonkin verran kokemusta ja tietoa ohjelmistotekniikoista sekä projektityöskentelystä. Jokaisella oli tarvittavaa teknistä ymmärrystä ja osaamista ohjelmistojärjestelmän toteuttamista varten. Monella oli myös työkokemusta ohjelmistoalalta. Yksikään kurssille osallistuneista opiskelijoista ei ollut työskennellyt Software Factoryssä edeltävässä opetusperiodissa, jolloin samalle asiakkaalle toteutettiin ensimmäinen projekti. Työympäristö ja kurssi olivat kuitenkin kahdelle kurssille osallistuneelle opiskelijalle ennestään tuttuja, sillä he olivat osallistuneet samalle kurssille jo kerran aiemminkin.

Ryhmän vetäjä (*coach*) oli ollut mukana myös edellisessä projektissa, joten hän oli työskennellyt asiakkaan kanssa jo aiemmin.

Asiakkaan puolelta projektiin osallistui kolme henkilöä. Yksi heistä oli yrityksen perustaja ja muut teknisempiä työntekijöitä. Kaikki olivat tarvittaessa projektitiimin käytettävissä ja tavoitettavissa tarpeen mukaan.

Ensimmäinen samalle asiakkaalle tehty projekti tehtiin 3.9.–19.10.2012. Projekti kesti myös kokonaiset 35 työpäivää. Projektiin osallistui myös viisi oppilasta, joista yksi päätti jättää kurssin kesken. Projektissa asiakkaalle kehitettiin selainpohjainen datan visualisointityökalu, jolla varsinaisesta pääpalvelusta kerättyä mittausdataa voitiin esittää graafisesti ja analysoida. Järjestelmä oli toteutettu käyttäen Javaa ja Spring Framework -kehystä. Ohjelmiston vaatimusten kannalta ensimmäisessä projektissa havaittiin muutamia ongelmia palvelun laatuun liittyvissä ei-toiminnallisissa vaatimuksissa. Muutamat vaatimuksista otettiin harkintaan vasta projektin loppupuolella, erityisesti tietoturva sekä suorituskyky. Suorituskyvyn kanssa havaittiin vasta projektin loppupuolella ongelmia datamäärien kasvaessa. Myös ohjelmistoon liittyvän dokumentaation laatu ei ollut täysin halutulla tasolla.

Tapaustutkimuksen kohteena olevan projektin aikana asiakkaalle toteutettiin suorituskykytestausohjelmisto, jonka avulla pyrittiin mittaamaan heidän nykyisen järjestelmänsä suorituskykyä ja selvittämään sen mahdollisia pullonkauloja. Suorituskykytestausohjelmistolla oli tarkoitus mitata palvelun metriikkadatan vastaanottavan puolen suorituskykyä ja skaalautuvuutta erilaisissa tilanteissa. Tämä metriikkadata on siis samaa, jota edellä mainitussa edellisessä projektissa visualisoitiin. Kehitystiimi kykeni tuottamallaan suorituskykytestausohjelmistolla arvioimaan ja mittaamaan kohdejärjestelmän kestämää kuormaa ja yhtäaikaisten yhteyksien määrää. Projektin aikana asiakkaalle rakennettiin myös prototyyppi nopeammasta metriikkadataa vastaanottavasta järjestelmän osasta. Prototyypin parempi suorituskyky voitiin varmistaa em. suorituskykytestausohjelmistolla. Lopuksi koostettiin myös lista suosituksista, joita suorituskykytestauksen tai muun työn myötä

nousi esille.

Projektissa käytettiin seuraavia teknologioita, työkaluja ja metodeja: Kanban, Git, Node.js, Express, Api-Easy, Vows, Backbone.js, jQuery, Underscore, Bootstrap, Jasmine, HighCharts, Mongoose, MongoDB, Mongoengine, PyMongo, Python 3.2, SNMP, PySNMP, MySQL.

6.1.2 Projektin vaiheet ja työskentelyprosessi

Kuvassa 6.1 esitetään projektin aikataulu, johon on kuvattu erilaiset loogiset osuudet ja niiden kestot. Kuva on koostettu vertaamalla kirjoittajan muistiinpanoja muuhun aineistoon.

Projektitiimi kokoontui asiakkaan kanssa yhteensä seitsemän kertaa. Kokoontumiset pidettiin aina Software Factoryn tiloissa. Osa uuden projektin tiimistä oli paikalla myös samalle asiakkaalle tehdyn edellisen projektin lopputapaamisessa. Tapaamisessa käytiin läpi edellisen projektin onnistumisia ja oppeja seuraavaa projektia varten sekä suunniteltiin tulevaa eli tämän tapaustutkimuksen projektia.

Tämän tapaustutkimuksen projektin ensimmäisen vaiheen tavoitteet ja laajuus määriteltiin ensimmäisen kahden viikon aikana. Asiakkaalla oli jo ennen projektin alkua ehdotuksia ja ideoita projektin tulevaksi sisällöksi. Käytettävissä olevasta ajasta johtuen aihetta rajattiin, jotta projektin aikana olisi mahdollista toimittaa valmis ohjelmisto. Ensimmäisessä vaiheessa päätettiin toteuttaa suorituskykytestausohjelmisto, jonka avulla olemassa olevan järjestelmän suorituskykyä olisi mahdollista arvioida. Tämä mahdollisti suorituskykytestausohjelmiston hyödyntämisen myöhemmin, kun kehitettiin suorituskykyisempää prototyyppiä, analysointiin olemassa olevan järjestelmän suorituskykyä sekä laadittiin järjestelmään liittyviä parannusehdotuksia.

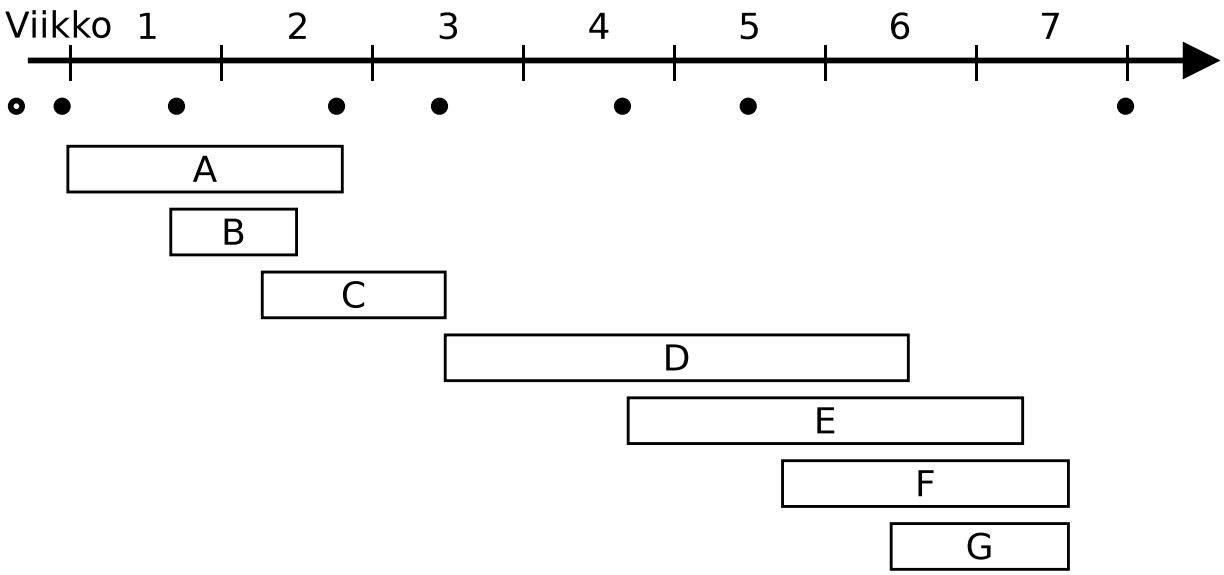
Teknologiavaihtoehtoja ryhdyttiin evaluoimaan ensimmäisen viikon lopussa. Tiimi selvitti, minkälaisia vaihtoehtoisia komponentteja toteutettavaan järjestelmään voitaisiin valita ja millaisia seurauksia näillä valinnoilla olisi järjestelmän arkkitehtuuriin. Tällä pyrittiin saavuttamaan hallittava ja järkevä alustava arkkitehtuuri ennen koodaamisen aloittamista. Tiimi esitteli selvityksen asiakkaalle ja sai hyväksynnän suunnitelmalleen ja valinnoilleen. Asiakas antoi tiimille luvan tehdä jatkossa itsenäisesti perusteltuja päätöksiä teknologioista.

Suorituskykytestausohjelmiston ensimmäisen version toteutus alkoi valittujen teknologioiden pohjalta toisen viikon aikana. Tänä aikana myös yksi tiimin jäsenistä päätti lopettaa

kurssin. Työmäärää tai projektin laajuutta ei kuitenkaan tästä syystä muutettu, sillä sen ei koettu aiheuttavan ongelmia. Kehitystiimi ei toisaalta myöskään samalla viikolla pidetyssä palaverissa halunnut lähteä suunnittelemaan ja toteuttamaan mitään uusia kokonaisuuksia testausohjelmiston vaatiman työmäärän epävarmuudesta johtuen. Seuraavalla viikolla ohjelmiston toimivaa ensimmäistä versiota esiteltiin demotilaisuudessa asiakkaalle. Asiakas hyväksyi tapaamisessa ohjelmiston kehityksen viemisen loppuun toimintoja laajentamalla. Tämä tarkoitti mittaustavan parantamista, uusien suorituskykymetriikoiden lisäämistä sekä myös parannuksia suorituskyvyn mittauksien esittämiseen. Tapaamisessa päätettiin myös, että seuraavaksi tiimi ryhtyy suunnittelemaan asiakkaalle uutta kestävää arkkitehtuuria metriikkadatan vastaanottavaan osaan ja mahdollisesti ajan puitteissa myös toteuttamaan tätä arkkitehtuuria.

Projektin loppuajan tavoitteet määriteltiin neljännellä viikolla asiakastapaamisessa. Edellisessä samalle asiakkaalle tehdyssä projektissa dokumentaation laatu oli aiheuttanut ongelmia, joten nyt siihen haluttiin kiinnittää erityistä huomiota ennen projektin päättymistä. Projektin sekä suorituskykytestausohjelmiston dokumentaation ja asennusohjeen lisäksi päätettiin koostaa analyysiraportti. Raportissa tuli esittää analyysi nykyisen järjestelmän suorituskyvystä, sen pullonkauloista ja mahdollisista parannusehdotuksista. Parannusehdotuksien lisäksi päätettiin asiakkaan vaatimuksesta toteuttaa prototyyppi suorituskykyisemmästä metriikkadataa vastaanottavasta järjestelmän osasta, sillä alkuperäisen suunnitelman mukainen testiohjelmisto näytti valmistuvan hyvissä ajoin. Prototyypillä haluttiin erityisesti osoittaa ja varmistaa, että on mahdollista tuottaa sellainen järjestelmä, jossa samoja suorituskykyongelmia ei ole havaittavissa. Suorituskyky oli kriittinen vaatimus, sillä ruuhkatilanteessa ylimääräiset viestit tippuivat vääristäen palvelun keräämää statistiikkaa. Järjestelmässä ei ollut varsinaista keinoa varmistaa statistiikkaviestin saapuminen perille.

Dokumentaation ja testauksen analyysin kirjoittaminen aloitettiin viidennellä viikolla. Seuraavalla viikolla keskityttiin töiden viimeistelyyn, virheiden korjaamiseen ja refaktorointiin. Projektin viimeisen viikon alussa tehtiin viimeiset koodimuutokset ja paketoitiin projekti asiakkaalle luovutusta varten. Projektin tuotokset, suorituskykytestausohjelmisto dokumentaatioineen, analyysiraportti sekä prototyyppi, toimitettiin asiakkaalle viimeisellä viikolla. Projekti päättyi lopputapaamisessa suoritettuun projektin läpikäyntiin.



Kuva 6.1: Projektin aikajana kirjoittajan muistiinpanojen perusteella muuhun aineistoon vertailemalla.
A: projektin laajuuden ja tavoitteiden spesifiointi
B: teknologioiden evaluointi
C: suorituskykytestausohjelmiston ensimmäisen toimivan version toteutus
D: suorituskykytestausohjelmiston toteuttaminen loppuun
E: metriikkadatan vastaanottavan rajapinnan prototyyppi
F: dokumentaation ja analyysin kirjoittaminen
G: virheiden korjausta ja refaktorointia
●: asiakastapaaminen
○: edellisen projektin lopputapaaminen

Projektissa hiljaisen tiedon osuus vaatimuksista oli suuri. Päätason vaatimuksia ei erikseen kirjattu ylös, vaan projektin todelliset vaatimukset olivat pikemminkin asiakkaan visioita ja tavoitteita. Kehitystiimi johti näistä päätason vaatimuksista tarkempia vaatimuksia kommunikoimalla asiakkaan kanssa niin kasvokkain kuin pikaviestimien ja sähköpostin välityksellä. Tästä kehitystiimin ja asiakkaan välisen vuorovaikutuksesta syntyneestä yhteisymmärryksestä vaatimuksiin kehitystiimi sitten johti ja kirjasi tehtäviä Kanban-taululle. Kanban-taulun tehtäviä ei siis luotu erikseen kirjatuista vaatimuksista tai käyttäjätarinoista.

Projektin alussa asiakkaalla oli kaksi aihetta tapaustutkimuksen projektille. Edellisen projektin tuotoksen integroiminen osaksi tuotantojärjestelmää päätettiin jättää vanhan tiimin jatkokehitystehtäväksi, joten uuden projektin aiheeksi valikoitui "simulaattorin / suorituskykytesterin kehittäminen". Asiakkaan idea suorituskykytestausohjelman kehittämisestä nousi tarpeesta varmistaa, että liiketoiminnalle (asiakkaalle ja operaattoreille) tärkeä met-

riikkadata saadaan varmasti tallennettua ilman, että sitä pääsee katoamaan suorituskykyongelmien takia.

Asiakas halusi työkalun, jolla nykyjärjestelmän suorituskyky voidaan testata ja varmistaa. Tämä oli projektin kannalta kriittisin prioriteetti. Myöhemmin asiakkaan kanssa käydyissä keskusteluissa nousi esille vaatimus parantaa testattavan järjestelmänsä suorituskykyä. Tämä vaatimus vaihtui ja muuntui myöhemmin ensin uuden arkkitehtuurin suunnitteluun sekä toteutukseen ja lopulta analyysiraportin kirjoittamiseen sekä suorituskykyisemmän prototyypin tuottamiseen.

Muutokset vaatimuksiin tapahtuivat asiakkaan ja kehitystiimin välisen kommunikaation seurauksena. Projektin jäljellä olevat resurssit (käytännössä aika) ja niiden järkevä käyttö ohjasivat näitä muutoksia. Olemassa olevan järjestelmän parantamisen ja uuden arkkitehtuurin suunnittelun sekä toteuttamisen arvioitiin molempien kuluttavan suuren osan jäljellä olevista kehitysresursseista tutustumiseen, opetteluun ja suunnitteluun. Vaatimuksia muokattiin siten, että tiimin oli mahdollista projektin aikana tehdä valmiiksi jotakin asiakkaalle arvoa tuottavaa. Aikataulukaaviosta 6.1 nähdään tarkemmin, missä vaiheessa mitkäkin kokonaisuudet projektin aikana tuotettiin ja mikä niiden prioriteetti oli.

Kanban-taulun rakenne määriteltiin heti projektin ensimmäisen viikon alussa. Yksi projektihuoneen valkotauluista jaettiin viiteen sarakkeeseen, jotka kuvasivat tehtävän etenemisen eri vaiheita. Osat nimettiin vasemmalta alkaen: *todo*, *analyze*, *ongoing*, *review* ja *done*. Sarakkeille määriteltiin myös ylärajat sille, kuinka monta yhtäaikaista työtehtävää yksittäisessä sarakkeessa saa enintään kerralla olla lukuun ottamatta taulun aloitus- ja valmistumisvaihetta (*todo* ja *done*). Nämä rajoitteet on kerrottu tarkemmin taulukossa 6.1. Kanban-taulun rakenne säilyi muuttumattomana läpi koko projektin eikä siihen liittyviä vaiheita, vaiheiden rajoitteita tai vaihesiirtymien sääntöjä muokattu.

Todo-lokerossa olivat aloittamattomat ja analysoimattomat tehtävät. Tähän sarakkeeseen lisättiin kaikki uudet projektiin ajatellut työtehtävät. Sarakkeessa oleva työjono oli jossain määrin prioriteetin mukaisessa järjestyksessä. Järjestelmän rakenteesta, tehtävien riippuvuuksista ja ihmisten mieltymyksistä johtuen korkeimmalla ollut työtehtävä ei kaikissa tilanteissa kuitenkaan ollut se, jonka yksittäinen kehittäjä otti seuraavaksi työn alle työjonosta. Osa tässä lokerossa olevista tehtävistä karsiutui pois jo ennen *analyze*-vaihetta.

Analysointivaiheessa tehtävälle kirjoitettiin *definition of done*. Tässä vaiheessa selvitettiin myös, mitä kyseinen tehtävä tarkoittaa nykyisessä kontekstissaan ja projektissa tällä hetkellä, jolloin tehtävä saattoi muuttua hieman alkuperäisestä. Tehtävälle kirjattiin samaan fyysiseen lappuun liitteeksi ehto, jonka täytettyään kyseinen tehtävä voitiin katsoa valmis-

tuneeksi. Tätä ehtoa vasten oli mahdollista arvioida tehtävän tilaa *review*-vaiheessa. Ehdon kirjaamisen jälkeen tehtävä oli mahdollista siirtää *ongoing*-sarakkeeseen. Analysointivaiheessa oli myös mahdollista, että yksittäinen tehtävä jakautui useammaksi tehtäväksi, yhdistyi toisen tehtävän kanssa tai hylättiin kokonaan.

Ongoing-vaiheessa tehtävä pyrittiin suorittamaan siten, että sille aiemmin määritelty valmistumisehto täyttyisi. Tehtävä oli mahdollista siirtää *review*-vaiheeseen, kun tehtävää suorittanut henkilö oli mielestään saanut täytettyä tehtävälle määritellyn valmistumisehdon.

Review-vaiheessa pyrittiin varmistumaan siitä, että tehtävä oli suoritettu oikein. Tehtävän suorittamisen lopputulosta (eli yleensä tuotettua ohjelmakoodia) verrattiin tehtävässä määriteltyyn valmistumisehtoon. Samalla pyrittiin varmistamaan, että tehtävän suorittaminen täyttää (subjektiiviset) laatuvaatimukset eivätkä esimerkiksi versionhallintaan tallennettavat muutokset aiheuta odottamattomia virheitä ohjelmistoon. Tehtävä siirrettiin viimeiseen valmistuneiden tehtävien lokeroon (*done*), jos tarkastusta suorittaneen henkilön mielestä tehtävän suoritus täytti sille asetetut vaatimukset. *Review*-vaiheen ohjeena oli myös, että sama henkilö ei sekä suorittaisi tehtävää että arvioisi sen valmistumisehtojen täyttymistä. Aivan kaikkien työtehtävien osalta näin ei tapahtunut.

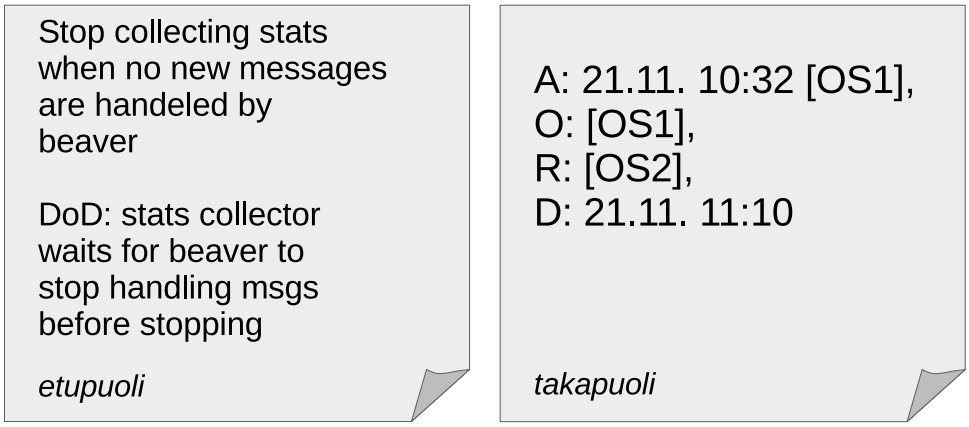
Done-sarakkeessa säilytettiin kaikki valmistuneet tehtävät. Sarake tyhjennettiin yleensä asiakkaalle pidettyjen katselmointipalavereiden yhteydessä. Tällä tavalla asiakkaalle kyettiin esittämään valmiit tehtävät, jotka olivat valmistuneet edellisen katselmoinnin jälkeen.

Kanban-taulun sarake	**Rajoitus** (work in progress)
Todo	
Analyze	3
Ongoing	4
Review	3
Done	

Taulukko 6.1: Projektin aikana käytetty Kanban-taulun rakenne ja rajoitukset

Kanban-taululle lisättäviin tehtäviin kirjattiin lyhyt kuvaus ominaisuudesta, ehdosta tai suoritettavasta tehtävästä. Kun tehtävä otettiin työn alle, tehtävälle määriteltiin ja kirjattiin analyysivaiheessa *definition of done*. Taululle lisättävä tehtävä sai olla minkälainen projektiin liittyvä suorite tahansa, kunhan sille voitiin määritellä edellä mainittu valmistumisehto. Lapun taakse kirjattiin luvussa esitetyt 5.2 tutkimusta varten kerättävät tiedot.

Esimerkkejä Kanban-taulun läpikulkeneista tehtävälapuista voi katsoa kuvista 5.1 ja 6.2.



Kuva 6.2: Esimerkki Kanban-taulun läpi kulkeneesta vaatimuksesta, joka liittyy mittaustuloksia keräävään osaan. Tekijöiden nimet on koodattu *OS1* ja *OS2*.

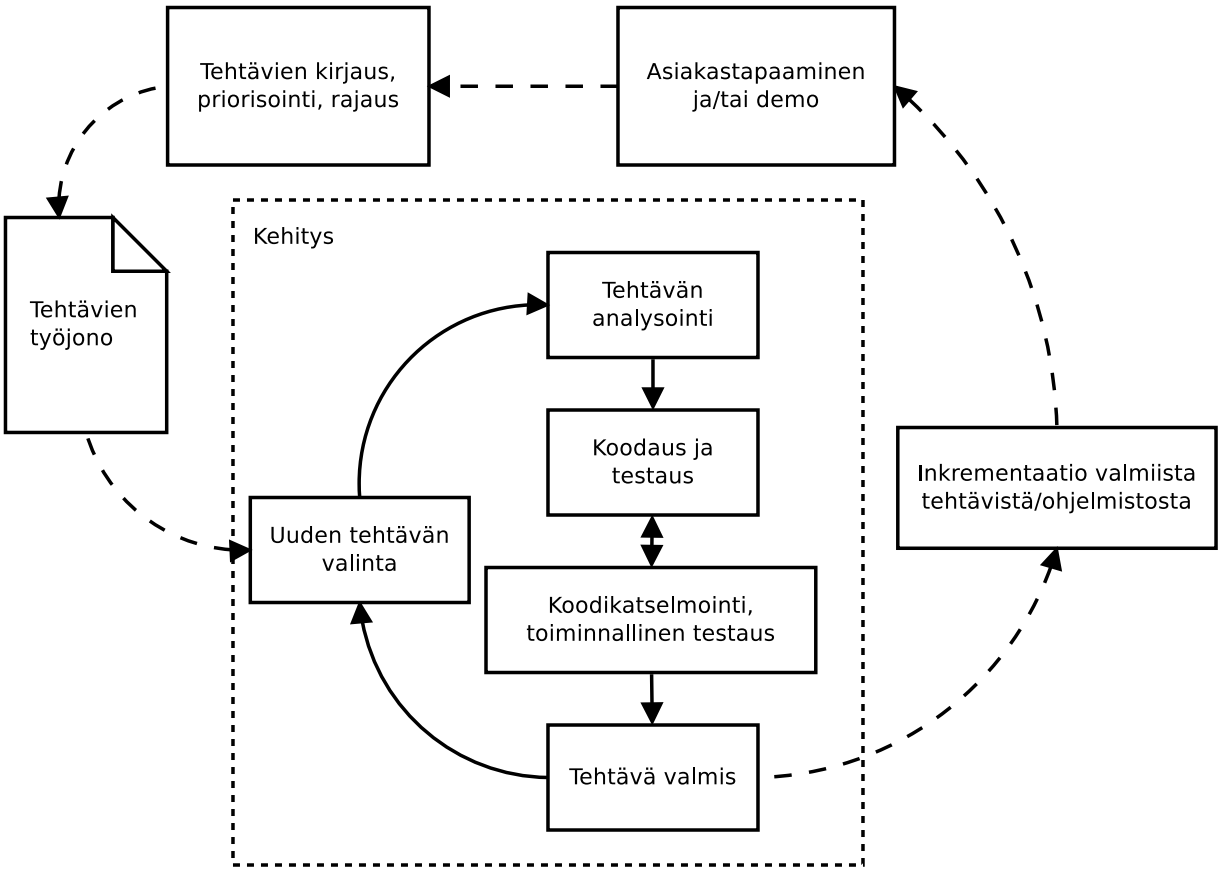
Prosessi, jolla projektia ohjattiin ja vietiin eteenpäin, määriteltiin ensimmäisen viikon aikana. Fyysistä Kanban-taulua lukuun ottamatta varsinaista työskentelyprosessia ei erikseen kirjattu mihinkään ylös, vaan sen pääperiaatteista keskusteltiin kehitystiimin kanssa projektin alussa. Muistiinpanojen pohjalta rakennettu kuvaus projektin työskentelyprosessista on esitetty kuvassa 6.3. Prosessi säilyi sen määrittelyn jälkeen muuttumattomana läpi koko projektin. Tiimi ei pitänyt projektin aikana erillisiä retrospektiivejä. Työskentelyprosessin parantamiseen tähtääviä kokouksia päätettiin pitää erikseen, jos sellaiselle nähtäisiin aihetta.

Projektin aikana tiimi kokoontui asiakkaan kanssa seitsemän kertaa. Tapaamiskerrat on merkitty kuvaan 6.1. Toisesta viikosta alkaen tapaamisissa katselmoitiin uusia valmistuneita tehtäviä. Lisäksi tapaamisissa kerättiin lisätietoa työjonossa olevia tehtäviä varten, priorisoitiin ja määriteltiin projektin seuraavia vaiheita sekä rajattiin tarvittaessa projektin laajuutta. Asiakkaan kanssa kommunikoitiin asiakastapaamisten lisäksi pikaviestimiä ja sähköpostia käyttäen.

Tapaamisessa keskusteltujen asioiden pohjalta tiimi lisäsi työjonoon tehtäviä palaverin yhteydessä, mutta myös palaverin jälkeen kehityksen aikana. Suuria linjauksia ei kirjattu erikseen, vaan ne olivat osittain hiljaista jaettua tietoa, siitä mikä projektissa on tärkeää. Yksittäisiä työtehtäviä ei priorisoitu asiakkaan toimesta, vaan tiimi teki sen itsenäisesti. Projektin isommat suuntaviivat, päätökset ja prioriteetit olivat asiakkaan päätettävissä, kuten esimerkiksi päätös prototyypin kehittämisestä.

Katselmoitavien ohjelmiston iteraatioiden sisältöjä ei sovittu etukäteen eli tapaamisissa ei sitouduttu saamaan tiettyjä tehtäviä valmiiksi ennalta määritellyssä ajassa. Tärkeämpää oli priorisoida kokonaisuuksia ja rajata projektia siten, että kaikki tärkeä saadaan valmiiksi. Tapaamisille oli sovittu alustavat päivämäärät, mutta niistä voitiin poiketa tai jättää ne pitämättä.

Tiimi piti jokaisen päivän aluksi Scrum-prosessin mukaisen *daily scrum* -palaverin, jossa kukin kävi läpi: mitä oli tehnyt eilen, mitä aikoi tehdä tänään ja oliko ongelmia, jotka estivät työn etenemistä (katso luku 4.2.2). Tarkemmat keskustelut, työskentelyä estävien asioiden ja ongelmien ratkaisu käytiin läpi palaverin jälkeen kyseiseen asiaan liittyvien henkilöiden kesken.



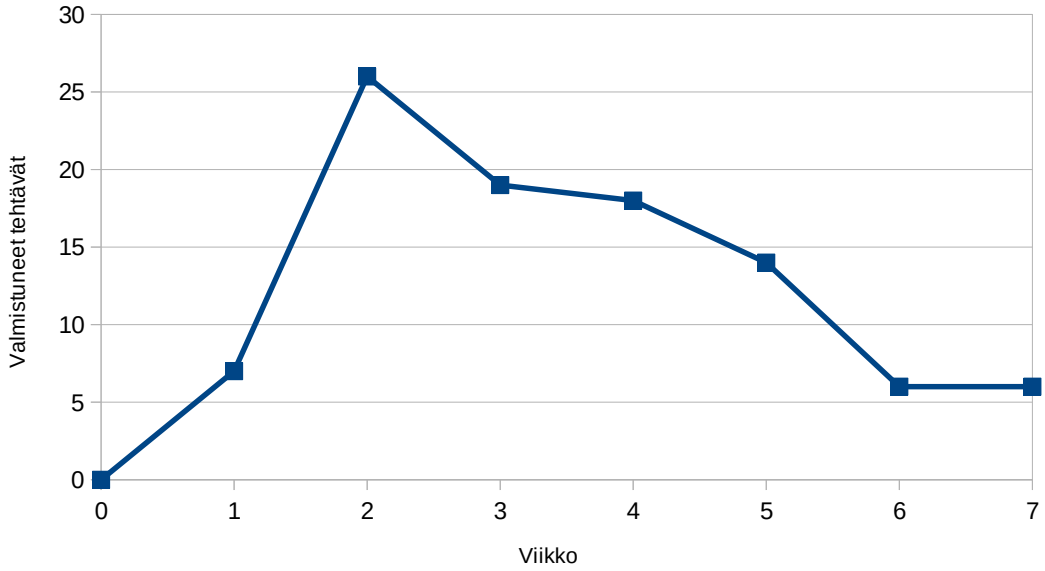
Kuva 6.3: Kirjoittajan muistiinpanojen perusteella muodostettu käsitys projektin prosessin vaiheista.

Uudet työtehtävät valittiin prioriteetin mukaisesti työjonosta. Kehittäjät ilmaisivat valitun tehtävän asettamalla kyseistä tehtävää kuvaavan lapun päälle oman taulumagneettinsa. Kehittäjän tuli myös katselmoida muiden kehittäjien tekemiä tehtäviä, jos niitä odotti Kanban-taulun *review*-sarakkeessa. Tehtävän valitsemisen jälkeen kehittäjä määrit-

teli tehtävälle sen valmistumisehdon, minkä jälkeen itsenäisesti tai toisen kehittäjä kanssa kirjoitti tehtävään vaadittavan koodin. Toinen tiimin jäsen katselmoi ja testasi valmiin tehtävän. Työ palautui alkuperäiselle kehittäjälle korjattavaksi, jos katselmoijan mielestä valmistumisehdon täyttymisessä tai työn laadussa oli korjattavaa. Valmiin tehtävän koodimuutokset yhdistettiin Kanban-prosessin *review*-vaiheen lopussa osaksi versionhallinnan pääkehityshaaraa, minkä jälkeen tehtävä kokonaisuudessaan katsottiin valmiiksi. Kuvassa 6.3 on myös erikseen kuvattu kehitysvaiheen sisäinen prosessi. Kehitystiimin jäsenillä ei ollut määrättyä etukäteisroolitusta. Suoritettujen tehtävien perusteella on kuitenkin havaittavissa, että kehitystiimin jäsenet ottivat itsenäisesti vastuulleen tiettyjä osia rakennettavasta järjestelmästä ja suorittivat paljon siihen liittyviä tehtäviä.

Projektin aikana valmistui yhteensä 98 tehtävää. Kuvassa 6.4 esitetään projektin tietyn viikon aikana valmistuneiden tehtävien määrä. Suuri piikki nähdään toisella viikolla, jolloin valmistui suurin osa projektin tehtävistä, jotka liittyivät projektin perusasioiden saattamiseen kuntoon. Näitä olivat muun muassa kehitys- ja testausympäristön pystyttäminen sekä teknologioiden kartoittaminen ja evaluoiminen. Kolmannesta viikosta alkaen tiimi pääasiassa toteutti ohjelmiston keskeisiä ominaisuuksia. Projektin loppupuolella tiettyjen tehtävien työmäärä oli alkupään tehtäviä suurempi. Esimerkiksi projektin loppupuolella valmistuneet dokumentointiin, analysointiin ja refaktorointiin liittyvät tehtävät olivat huomattavasti suuritöisempiä.

Kaikki työjonon tehtävistä eivät liittyneet suoraan kehitettävään ohjelmistoon. Mukana oli myös esimerkiksi projektinhallintaan liittyviä tehtäviä kuten *create calendar* ja *plan meeting*. Näillekin tehtäville pystyttiin kuitenkin helposti määrittelemään tilanteeseen sopiva valmistumisehto, joten ne lisättiin työjonoon.



Kuva 6.4: Projektin aikana valmistuneet tehtävät per viikko

Muutamissa Kanban-taulun läpi virranneissa post-it-lapuissa oli puutteellisesti kirjattu päättymisaika. Tehtävien luonteen ja käytössä olleen prosessin vuoksi nämä puutteelliset tiedot oli jälkikäteen muutamassa tapauksessa mahdollista täydentää analyysiä varten Git-versionhallinnan perusteella.

6.2 Ketterän vaatimusmäärittelyn havaitut hyödyt ja haasteet

Tässä luvussa käydään läpi miten ketterään vaatimusmäärittelyyn liittyvässä tutkimuskirjallisuudessa esitetyt hyödyt sekä haasteet (katso luku 4.3.2) olivat havaittavissa tapaustutkimuksen kohteena olevassa projektissa. Hyödyt ja haasteet käydään erikseen läpi omissa alaluvuissaan. Luvussa 6.3 esitetään vielä lyhyt yhteenveto tämän luvun havainnoista.

6.2.1 Hyödyt

Tässä luvussa esitetään analyysin tulokset siitä, miten ja miksi luvussa 4.3.2 mainitut ketterän vaatimusmäärittelyn hyödyt ilmenivät tapaustutkimuksen projektissa. Projektissa oli mahdollista havaita ja tunnistaa kaikki hyödyt.

Vähemmän työtä prosessin vuoksi

Projektissa oli käytössä hyvin kevyt prosessi, joka perustui suurelta osin kommunikaatioon kehitystiimin ja asiakkaan välillä sekä hiljaiseen tietoon. Vaatimusten määrittely, tarkentaminen, priorisointi ja projektin laajuuden rajaaminen tehtiin kommunikoimalla kehitystiimin ja asiakkaan kesken. Kehitystiimi keskusteli asiakkaan kanssa vaatimuksista tai pikemminkin asiakkaan visioista ja tavoitteista, jotka tämän vuorovaikutuksen ansiosta jalostuivat yhteiseksi jaetuksi tiedoksi projektin vaatimuksista ja prioriteeteista. Näitä vaatimuksia ei erikseen kirjattu ylös.

Projektin vaatimusten perusteella kehitystiimi lisäsi Kanban-taululle työtehtäviä. Tehtävien sisältö määriteltiin tarkemmin analyysivaiheessa, jossa ne kirjoitettiin siihen muotoon, että ne olivat yksiselitteisesti tehtävissä ja varmennettavissa tuotetusta ohjelmakoodista. Ohjelmakoodin, asennusohjeen ja tuotetun analyysiraportin lisäksi ainoa tuotettu dokumentaatio projektista ja sen vaatimuksiin liittyvistä tehtävistä olivat Kanban-taulun läpi kulkeneet post-it-laput. Yksittäiset tehtävät olivat ainut pakollinen prosessin tuottama dokumentaatio.

Perinteisiin menetelmiin verrattuna projektin prosessi sisälsi vähän prosessiin spesifisesti liittyvää työtä. Ketterän vaatimusmäärittelyn hyöty vähentyneestä prosessin vuoksi tehtävästä työstä oli täten mahdollista havaita projektissa.

Parantunut ymmärrys vaatimuksista

Projektin alussa kaikki sen vaatimukset eivät olleet kattavasti selvillä. Projektin idea suorituskykytestausohjelman rakentamisesta oli tiedossa, mutta alussa ei ollut täysin selvää, mitä tuli mitata ja miten. Vaatimusten priorisointia, määrittelyä, tarkentamista ja projektin laajuuden muokkaamista tapahtui läpi koko projektin kehitystiimin ja asiakkaan välisen kommunikaation kautta. Asiakas ei ollut aina läsnä projektitilassa, mutta oli kuitenkin helposti tavoitettavissa, ja tapaamisten järjestäminen oli mahdollista lyhyellä varoitusajalla. Kasvokkain tapahtuneiden tapaamisten yhteydessä kehitystiimi ja asiakas kävivät läpi tuleviin tehtäviin liittyviä kysymyksiä ja määrittelivät tarkemmin epäselviä vaatimuksia. Kehitystiimi sai myös palautetta valmistuneista tehtävistä, jolla validoitiin jo valmiita ominaisuuksia.

Suorituskykyohjelmiston edetessä pidemmälle myös ymmärrys mitattavista muuttujista ja oikeanlaisesta mittaustavasta selveni. Tarpeellisia muuttujia lisättiin ja mittaustapaa muokattiin simuloimaan oletetun kaltaista verkkoliikennettä paremmin. Myös asiakkaan

tarve parantaa nykyjärjestelmänsä suorituskykyä testausohjelman avulla ymmärrettiin ja jäljellä olevia kehitysresursseja käytettiin siten, että saatiin tuotettua asiakkaalle välineitä ja tietoa suorituskyvyn parantamiseen.

Suora yhteys asiakkaaseen mahdollisti vaatimusten määrittelyn, priorisoinnin ja validoinnin suoran kommunikaation kautta. Hyöty parantuneesta vaatimusten ymmärtämisestä oli täten mahdollista havaita projektissa.

Vähentynyt ylikuormittaminen

Kehitystiimin ja asiakkaan oli mahdollista priorisoida suorituskykytestausohjelmiston valmistuminen etusijalle. Ohjelmiston jäljellä olevan työmäärän epävarmuudesta johtuen muita kokonaisuuksia ei aloitettu ennen työmäärän tarkentumista. Kehitystiimi ja asiakas halusivat ensin varmuuden siitä, että tärkein työ on mahdollista saada valmiiksi, ja että tiimillä oli todella resursseja toteuttaa muitakin kokonaisuuksia projektissa.

Kun testausohjelman työmäärän oli tarkentunut, muita asiakkaan ideoita kyettiin ottamaan työn alle. Projektin laajuutta, tavoitteita ja vaatimuksia muokattiin ja niiden osalta huomioitiin käytössä oleva aika ja kehitysresurssit. Testausohjelman avulla oli tarkoitus parantaa olemassa olevan järjestelmän suorituskykyä tekemällä siihen tarvittavia muutoksia. Jäljellä oleviin resursseihin suhteutettuna päädyttiin lopulta kuitenkin vain analysoimaan nykyjärjestelmää ja tuottamaan suorituskykyisempi prototyyppi.

Kehitysresurssien osalta pyrittiin välttämään niiden ylikuormittamista. Hyöty vähentyneestä ylikuormittamisesta oli täten mahdollista havaita projektissa.

Muutostarpeeseen vastaaminen

Suorituskykytestausohjelmiston osalta mitattavat muuttujat ja mittaustapa kehittyi läpi projektin. Projektin alussa ei ollut täysin selvää, mitä tuli mitata ja miten. Näkemys näistä kehittyi projektin aikana. Uusia mitattavia muuttujia lisättiin, kun niiden nähtiin tuottavan lisäarvoa mittaustuloksiin. Esimerkiksi onnistuneen lähetyksen lisäksi tarkasteltiin myös todellisuudessa vastaanotettujen viestien määrää ja palvelimen sekä Java-virtuaalikoneen resurssien käyttöä seurattiin. Varsinainen testausalgoritmi kehittyi myös vaiheittain paremmaksi mittaamaan testattavan järjestelmän suorituskyvyn raja-arvoja, muun muassa lisäämällä logiikkaa testattavan järjestelmän lämmittelyyn ja nostamalla siihen kohdistuvaa kuormaa vaiheittain kohti maksimaalista testiarvoa.

Testausohjelman avulla asiakas halusi, että tiimin ryhtyisi heti parantamaan testattavan järjestelmän suorituskykyä. Koska projektin alussa oli epävarmuutta keskeneräisen testausohjelmiston työmääräarviosta ja projektin jäljellä olevasta työajasta, tämä idea muuntui myöhemmin ensin uuden arkkitehtuurin suunnitteluun ja sen toteutuksen aloittamiseen. Lopulta suunnitelma muuttui analyysiraportin kirjoittamiseen ja suorituskykyisemmän prototyypin rakentamiseen. Näissä muutoksissa tärkeänä tekijänä oli halu käyttää jäljellä oleva järkevästi siten, että ei sitouduttu tekemään mitään, minkä koettiin todennäköisesti jäävän keskeneräiseksi, ja jossa suurin osa ajasta jouduttaisiin käyttämään asian tutustumiseen ja opetteluun. Tämän sijaan haluttiin tuottaa valmista tietoa ja ohjelmakoodia, josta olisi asiakkaalle hyötyä.

Projektin aikana pystyttiin vastaamaan muutostarpeisiin. Ketterän vaatimusmäärittelyn hyöty muutostarpeeseen vastaamisesta oli täten mahdollista havaita projektissa.

Nopeat toimitukset ja validaatio

Kehitystiimi esitteli toisesta viikosta alkaen lähes viikoittain tapaamisten yhteydessä ohjelmistoon sekä muuhun työhön liittyviä valmistuneita tehtäviä asiakkaalle. Kanban-taulun *done*-sarakkeessa olleet tehtävät käytiin läpi, ja asiakas antoi oman hyväksyntänsä tehtäville täten validoiden valmiit tehtävät. Varsinkin käyttäjälle (käyttöliittymässä) näkyviä uusia ominaisuuksia esiteltiin. Yhtään valmista tehtävää ei näiden kokousten aikana palautettu takaisin työn alle Kanban-taululle.

Asiakastapaamiset eivät olleet kuitenkaan pelkästään katselmointipalavereja, vaikka tapaamisten yhteydessä tarkasteltiin valmistuneita ominaisuuksia. Kehitystiimi keskusteli asiakkaan kanssa myös projektin statuksesta sekä määritteli vaatimuksia ja tulevia kokonaisuuksia.

Kaikki *done*-sarakkeessa olevat tehtävät olivat osa versionhallinnan pääkehityshaaraa. Tässä mielessä pääkehityshaara oli aina tuorein toimitettu versio ohjelmakoodista. Tätä koodia ei kuitenkaan lähetetty asiakkaalle tai asennettu asiakkaan testattavaksi mihinkään testiympäristöön projektin aikana. Sitä ei projektin aikana vaadittu eikä pyydetty. Mikään ei toisaalta olisi ollut sen esteenä. Asennusohje kirjoitettiin vasta projektin loppupuolella ja projektin tuotos luovutettiin asiakkaalle aivan projektin lopuksi.

Asiakas validoi valmiit tehtävät säännöllisten tapaamisten yhteydessä. Kehitystiimi integroi valmiit koodimuutokset osaksi pääkehityshaaraa, joka lopulta toimitettiin asiakkaalle. Hyöty nopeista toimituksista ja validaatiosta on täten havaittavissa projektissa.

Parantuneet asiakassuhteet

Asiakkaan ja kehitystiimin välillä ei ollut havaittavissa konfliktitilanteita tapaamisten yhteydessä tai muussa kommunikaatiossa. Vaatimuksista ja projektista keskustelu pysyi aina asiallisena. Myöskään projektin tavoitteiden ja projektin laajuuden pienentäminen ei aiheuttanut ristiriitatilanteita. Asiakas toimi tiimiä kohtaan jopa kannustavasti. Kurssin lopussa pidetyssä loppupalaverissa kehitystiimi sai yleisesti ottaen hyvää palautetta motivaatiosta, osaamisesta ja tiimihengestä.

Hyöty parantuneista asiakassuhteista on analyysin perusteella mahdollista havaita projektissa.

6.2.2 Haasteet

Tässä alaluvussa esitetään analyysin tulokset siitä, miten luvussa 4.3.2 mainitut ketterän vaatimusmäärittelyn haasteet ilmenivät tapaustutkimuksen projektissa ja toisaalta myös siitä, miksi jotkin haasteista eivät olleet havaittavissa projektissa. Haasteiden osalta oli mahdollista havaita ja tunnistaa ainoastaan epätarkkoihin työmääräarvioihin liittyvä haaste.

Ongelmat liittyen asiakkaaseen tai asiakasedustajiin

Kehitystiimillä ei projektissa ollut käytössään aina läsnä olevaa asiakasta, joka usein nähdään parhaana vaihtoehtona ja mahdollistajana asiakkaan ja kehitystiimin väliselle runsaalle kommunikaatiolle. Asiakas oli kuitenkin suoraan ja nopeasti tiimin tavoitettavissa ja käytettävissä pikaviestimien avulla. Tapaaminen kasvotusten oli tarvittaessa mahdollista järjestää yleensä jo seuraavalle päivälle. Yhtä viikkoa lukuun ottamatta kehitystiimi tapasi asiakkaan kanssa vähintään kerran viikossa, jolloin katselmoitiin valmistuneet tehtävät ja ohjelmiston uudet ominaisuudet. Myös projektin status käytiin läpi ja tarkennettiin projektin nykyisiä ja uusia vaatimuksia.

Asiakas siirsi tuoteomistajan vastuuta suorituskykytestiohjelmasta kehitystiimille, sillä asiakkaan mukaan kehitystiimi tunsi uudet teknologiat paremmin ja asiakkaan mielestä näytti siltä, että kehitystiimi tiesi mitä oli tekemässä. Tapaamisissa asiakas seurasi testiohjelman edistymistä ja validoi siihen rakennetut uudet ominaisuudet.

Analyysin perusteella projektissa ei voitu havaita haastetta, joka olisi johtunut asiakkaaseen liittyvistä ongelmista.

Käyttäjätarinoiden riittämättömyys

Hiljaisen tiedon osuus projektin vaatimuksista oli suuri. Päätason vaatimuksia ei projektissa erikseen kirjattu ylös, vaan ne muodostuivat asiakkaan visioista ja tavoitteista. Kehitystiimin ja asiakkaan välisen kommunikaation avulla näistä visioista syntyi kehitystiimille käsitys projektiin liittyvistä vaatimuksista.

Vaatimusten keskinäistä johdonmukaisuutta, verifioitavuutta tai muitakaan laatukriteereitä on tapaustutkimuksen projektin vaatimusten kirjaamattoman luonteen vuoksi mahdotonta selvittää. Ainoa vaatimuksiin viittaava dokumentaatio on Kanban-taulun läpi virranneet tehtävät, jotka puolestaan eivät ole käyttäjätarinoiden muodossa. Ei-toiminnallisia vaatimuksia ei Kanban-taulun tehtäviin kirjattu tai huomioitu, joten nekin kuuluivat samaan kirjaamattomien vaatimusten joukkoon. Jäljitettäviä vaatimukset tai niistä johdetut Kanban-taulun tehtävät eivät olleet. Koodin ja tehtävän välinen yhteys ei ole suoraan pääteltävissä.

Analyysin perusteella voidaan kuitenkin todeta, että vaatimusten luonne ei projektin pituuden ja toteutettujen kokonaisuuksien yksinkertaisuuden vuoksi ehtinyt aiheuttaa havaittavia ongelmia. Kirjaamattomien käyttäjätarinoiden täydellisen puuttumisen vuoksi voidaan todeta analyysin osalta, että käyttäjätarinoiden riittämättömyyteen liittyvää haastetta ei havaittu. Samasta syystä on todettava myös, että tätä haastetta ei voitu myöskään kunnolla arvioida.

Haasteet vaatimusten priorisoinnissa

Projektin tärkeiden vaatimusten prioriteetit sovittiin suoran kommunikaation kautta asiakkaan ja kehitystiimin välillä. Tärkein prioriteetti projektissa oli saada suorituskykytestausohjelmaan liittyvät tehtävät valmiiksi. Testausohjelma oli oleellisessa osassa suhteessa kaikkiin muihin projektiin suunniteltuihin töihin. Tämä priorisoitiin siten etusijalle ja muut työt aloitettiin vasta, kun testiohjelma alkoi tuottaa kohdejärjestelmästä hyödyllisiä tuloksia, jotka indikoivat suorituskyvystä. Asiakas halusi projektin alkupuolella priorisoida myös nykyjärjestelmän suorituskyvyn parannuksia ja aloittaa niiden toteuttamista. Kehitystiimi ja asiakas pääsivät yhteisymmärrykseen siitä, että ilman toimivaa testausohjelmistoa ei suorituskykyä ja siinä mahdollisesti olevia pullonkauloja voinut kuin arvailla.

Yksittäisten Kanban-taulun tehtävien osalta tiimi määritti itse niiden prioriteetin ja toteutusjärjestyksen, eikä asiakas osallistunut kaiken työn priorisointiin. Projektin suuremmat tavoitteet ja vaatimukset olivat kuitenkin lopulta asiakkaan päätettävissä.

Analyysin pohjalta todetaan, että vaatimusten priorisoinnissa ei havaittu haasteita tässä projektissa.

Kasvava tekninen velka

Projektin alussa kehitystiimi selvitti, minkälaisia vaatimuksia etenkin suorituskyvyn kannalta itse suorituskykytestausohjelmalla tulisi olla. Tämän perusteella pyrittiin kartoittamaan teknologioita, jotka mahdollistaisivat nämä vaatimukset järjestelmälle. Tiimi evaluoi kartoittamansa eri vaihtoehdot ja varmisti niiden yhteensopivuuden. Kehitystiimi esitteli vaihtoehtoiset toteutustavat asiakkaalle ja yhteisesti päädyttiin valitsemaan sopivat teknologiat projektiin. Tällä erillisellä evaluoinnilla ja selvitystyöllä pyrittiin estämään se, että valittu arkkitehtuuri ja teknologiat osoittautuisivat projektin myöhemmässä vaiheessa riittämättömiksi.

Osalle ohjelmakoodista kirjoitettiin myös yksikkötestejä. Tiimi ei noudattanut tiukasti esimerkiksi testivetoista ohjelmistokehitystä (TDD), mutta testit kirjoitettiin aina kehitettävän ominaisuuden kanssa samanaikaisesti. Kehityksen aikana nämä testit auttoivat tunnistamaan rajapintoihin ja käyttöliittymään liittyviä rikkovia muutoksia.

Kehitystyön aikana ei ilmennyt sellaisia ongelmia, jotka olisivat johtuneet teknisestä velasta.

Tukeutuminen hiljaiseen tietoon vaatimuksista

Projektissa asiakkaan visio ja tavoitteet kommunikoitiin sanallisesti kehitystiimille. Vaatimukset ja niiden prioriteetit määriteltiin asiakkaan ja kehitystiimin välisen suoran kommunikaation avulla, mutta näistä varsinaisista vaatimuksia ei tuotettu mitään kirjallista dokumentaatiota. Hiljaisen tiedon osuus vaatimuksista oli siis suuri. Ainut prosessin tuottama dokumentaatio oli Kanban-taululle näistä kirjaamattomista vaatimuksista johdetut yksittäiset tehtävät. Valmiiden tehtävien katselmoinnin kautta myös ohjelmakoodi ilmensi valmiita ja validoituja vaatimuksia.

Projektin aikana yksi kehitystiimin jäsen päätti lopettaa kurssin kesken. Tämä tapahtui projektin alkupuolella tiimin vasta evaluoidessa sopivia teknologioita projektiin. Tiimi ei tässä kohtaa kokenut tarvetta tehdä muutoksia projektin laajuuteen tai työmäärään, sillä yhden henkilön lähtemisen ei koettu siinä tilanteessa aiheuttavan ongelmia.

Aineiston analyysin perusteella ei voida osoittaa, että hiljainen tieto olisi aiheuttanut

havaittavia haasteita projektissa.

Epätarkat työmääräarviot

Projektin todellisia vaatimuksia ei ollut kirjallisessa muodossa. Kanban-taulun työjonossa olleet tehtävät eivät olleet kattava kuvaus kaikista projektin tiedossa olleista vaatimuksista, vaan uusia tehtäviä kirjattiin vapaasti projektin aikana pitäen kuitenkin mielessä asiakkaan kanssa sovitut projektin prioriteetit. Toteutettavien kokonaisuuksien työmääriä ei arvioitu tarkasti eikä Kanban-taulun tehtäviin tehty myöskään työmääräarvioita. Kehitystiimin yksittäisen viikon aikana valmiiksi saamien tehtävien määrä oli korkeimmillaan toisella viikolla, jonka jälkeen viikoittain valmistuneiden tehtävien määrä laski tasaisesti loppua kohti (katso kuvaaja 6.4).

Epävarmuus toteutettavan testausohjelman vaatimasta työmäärästä johti siihen, että kehitystiimi ei projektin toisella viikolla pidetyssä palaverissa halunnut vielä sitoutua toteuttamaan mitään uusia kokonaisuuksia. Testausohjelmisto oli priorisoitu etusijalle, joten muiden töiden aloittaminen siirtyi, kunnes työmäärästä saatiin työn edistyessä parempi käsitys. Tämä ainakin osittain johti muutoksiin myöhempiin asiakkaan ideoimiin kokonaisuuksiin.

Analyysin perusteella epätarkkoihin työmääräarvioihin liittyvä haaste ilmeni tapaustutkimuksen projektissa.

6.3 Yhteenveto havaituista vaatimusmäärittelyn hyödyistä ja haasteista

Analyysin perusteella projektissa pystyttiin havaitsemaan ketterän vaatimusmäärittelyn raportoituja hyötyjä ja myös yksi siihen liittyvistä haasteista. Taulukoissa 6.2 ja 6.3 esitetään vielä lyhyt yhteenveto projektissa havaituista ketterän vaatimusmäärittelyn hyödyistä sekä haasteista perustuen luvussa 4.3.2 esitettyihin Heikkilä et al. (2015) ketterän vaatimusmäärittelyn hyötyihin ja haasteisiin.

Hyöty	**Havaittiin**
Vähemmän työtä prosessin vuoksi	Kyllä
Parantunut ymmärrys vaatimuksista	Kyllä
Vähentynyt ylikuormittaminen	Kyllä
Muutostarpeeseen vastaaminen	Kyllä
Nopeat toimitukset ja validaatio	Kyllä
Parantuneet asiakassuhteet	Kyllä

Taulukko 6.2: Yhteenveto ketterän vaatimusmäärittelyn havaituista hyödyistä projektissa

Haaste	**Havaittiin**
Ongelmat liittyen asiakkaaseen tai asiakasedustajiin	Ei
Käyttäjätarinoiden riittämättömyys	Ei*
Haasteet vaatimusten priorisoinnissa	Ei
Kasvava tekninen velka	Ei
Tukeutuminen hiljaiseen tietoon vaatimuksista	Ei
Epätarkat työmääräarviot	Kyllä

Taulukko 6.3: Yhteenveto ketterän vaatimusmäärittelyn havaituista haasteista projektissa

*Projektissa ei hyödynnetty käyttäjätarinoita vaatimusten kuvaamiseen.

7 Pohdinta

Tässä luvussa esitetään pohdinta liittyen tutkielman tuloksiin. Pohdinta keskittyy erityisesti tapaustutkimuksen tuloksissa havaittuihin ketterän vaatimusmäärittelyn hyötyihin ja haasteisiin, mutta siinä tehdään myös muutamia huomiota tutkimuskirjallisuuteen liittyen.

Ketterässä vaatimusmäärittelyssä toteutuvat perinteisen vaatimusmäärittelyn eri vaiheet eli vaatimusten esillesaanti, analyysi ja neuvottelu, dokumentointi, validointi sekä hallinta (Ramesh et al., 2010). Ketterästi tehdyn vaatimusmäärittelyn konkreettinen suoritustapa eroaa monelta osin perinteisestä tavasta tehdä vaatimusmäärittelyä. Tästä huolimatta näyttäisi, että kaikki perinteisen vaatimusmäärittelyn vaiheet suoritetaan tavalla tai toisella myös ketterissä ohjelmistoprojekteissa. Vaatimusmäärittelyn voidaan siis nähdä toteutuvan täysmittaisesti myös ketteriä menetelmiä ja siten ketterää vaatimusmäärittelyä hyödyntämällä. Vaatimusmäärittelyn toteutumisen kannalta voidaan siis olettaa ketterien menetelmien olevan varteenotettava vaihtoehto ohjelmistoprojektin prosessiksi. Esimerkiksi tässä tutkielmassa erikseen mainitut ketterät menetelmät XP ja Scrum näyttävät Lucia ja Qusef (2010) suorittaman vertailun perusteella toteuttavan kaikki vaatimusmäärittelyn vaiheet.

Ketterään vaatimusmäärittelyyn liitetyt hyödyt perinteiseen vaatimusmäärittelyyn verrattuna voidaan saavuttaa ketterällä menetelmällä toteutetussa ohjelmistoprojektissa. Hyödyt oli mahdollista havaita startup-yritykselle pienellä tiimillä toteutetussa yksinkertaisessa asiakasprojektissa. Vaatimukset eivät olleet projektin alussa etukäteen tarkkaan määriteltyjä, vaan ne kehittyivät ja saatiin esille projektin kuluessa samalla, kun ohjelmistokin kehittyi. Se, että asiakasprojektissakin voitiin havaita hyötyjä, voisi viitata siihen, että projektiin alun perin valittu ketterä lähestymistapa oli perusteltu, sillä vääränlaisella prosessilla hyötyjä ja haasteita oltaisiin voitu kohdata eri suhteessa.

Toisaalta Heikkilä et al. (2015) mukaan kun verrataan niitä hyötyjä, joita saavutetaan ketterällä vaatimusmäärittelyllä, perinteisiin ohjelmistokehitysmenetelmiin, voidaan havaita, että hyödyt ovat vastaavia ketterillä menetelmillä yleisesti saavutettavien hyötyjen kanssa. Ketterien menetelmien pyrkimys tuottaa arvoa mahdollisimman aikaisessa vaiheessa ja mahdollisuus reagoida muutoksiin liittyvät kuitenkin usein juuri vaatimuksiin, joten samankaltaiset hyödyt saattavat vain kertoa siitä, miten oleellinen osa vaatimusmäärittelyllä

on.

Suurinta osaa ketterään vaatimusmäärittelyyn liitettävistä haasteista ei kyetty tapaustutkimuksessa havaitsemaan. Ainoastaan epätarkkoihin työmääräarvioihin liittyvä haaste oli selvästi havaittavissa, sillä se vaikutti projektissa toteutettujen kokonaisuuksien aikatauluihin ja saattoi mahdollisesti myös osaltaan johtaa projektin vaatimusten muuttumiseen.

Asiakkaan ja kehitystiimin välisessä kommunikaatiossa ei esiintynyt sellaisia puutteita, jotka olisivat olleet havaittavissa. Asiakkaaseen liittyvät ongelmat olisivat todennäköisesti heijastuneet monelle eri osa-alueelle, jolloin useampia haasteita ja harvempia hyötyjä olisi saatettu havaita. Toisaalta ketterän prosessin ansiosta kehitystiimi olisi tällaisessa tilanteessa saattanut – ongelman huomatessaan – pyrkiä hakemaan siihen jonkinlaista ratkaisua.

Heikkilä et al. (2015) arvelivat, että osa haasteista nousee selvemmin esille kehitettäessä suuria ja monimutkaisia järjestelmiä. Tapaustutkimuksen projektissa tuotettu ohjelmisto oli suhteellisen pieni ja yksinkertainen. Myös tiimi oli pieni ja kykeni työskentelemään yhdessä koko projektin ajan samassa tilassa. Jos aineistona olisi ollut suurempi, pidempikestoisempi ja monimutkaisempi projekti, olisi siis esiin saattanut nousta useampia ketterään vaatimusmäärittelyyn liittyviä haasteita. Esimerkiksi asiakkaan saatavuudessa ja käyttäjätarinoiden riittämättömyydessä olisi saatettu havaita ongelmia, jos projektissa olisi työskennelty useamman tiimin voimin. Tutkimuksen toistaminen tämänlaisessa ympäristössä saattaisi tarjota enemmän mahdollisuuksia havaita ketterän vaatimusmäärittelyn haasteita.

Ketterän vaatimusmäärittelyn haastetta, joka liittyy käyttäjätarinoiden riittämättömyyteen, oli kerätystä aineistosta käytännössä mahdotonta analysoida, sillä projektissa ei käytetty käyttäjätarinoita kuvaamaan vaatimuksia. Suurin osa todellisista vaatimuksista oli pikemminkin asiakkaan visioita ja tavoitteita, jotka tiimi oli sisäistänyt kommunikaation kautta. Tapaustutkimuksen projektin kannalta olisi voinut olla mahdollista, että kirjatut ylemmän tason vaatimukset tai tavoitteet, esimerkiksi käyttäjätarinoiden muodossa, olisivat voineet helpottaa työn validointia, priorisointia ja työmäärän arviointia. Käyttäjätarinat olisivat voineet parantaa asiakastapaamisissa suoritettuja katselmointeja, sillä käyttäjätarinan perusteella asiakas olisi voinut helpommin varmentaa, onko yksiselitteinen käyttäjätarina toteutettavissa järjestelmällä ja onko sen toteutus asiakkaan tarpeita vastaava. Käyttäjätarinat olisivat myös voineet auttaa asiakasta priorisoimaan tärkeimpiä ominaisuuksia etusijalle ja toisaalta karsimaan turhia ominaisuuksia. Projektin vaatimuksia olisi voitu esimerkiksi suorituskykytestausohjelman kannalta kuvata seuraavanlaisilla

Lucassen et al. (2015) määrittelemän suosituksen mukaisilla käyttäjätarinoilla:

- *Ylläpitäjänä haluan nähdä, mikä on järjestelmän tarvitsema resurssien määrä tietyllä kuormalla.*
- *Ylläpitäjänä haluan selvittää, mikä on järjestelmän kestämä samanaikaisten yhteyksien määrä, jotta tiedän kestääkö järjestelmä sen oletetun käytön.*
- *Kehittäjänä haluan vertailla eri mittauskertojen tuloksia, jotta voin nähdä, miten tekemäni koodimuutokset vaikuttavat järjestelmän suorituskykyyn.*

Käyttäjätarinaformaatti yksinään olisi silti saattanut osoittautua riittämättömäksi kuvaamaan suorituskykyä mittaavaa algoritmia tai muita teknisiä ohjelmiston arkkitehtuuriin ja teknologiavaihtoehtoihin liittyviä vaatimuksia, joiden päälle monet edelläkin mainituista käyttäjätarinoista olisivat rakentuneet.

Analyysin pohjana ollut aineisto ei välttämättä tarjonnut kaikissa tilanteissa riittävästi mahdollisuuksia arvioida ketterään vaatimusmäärittelyyn liittyvien haasteiden esiintymistä projektissa. Suuri osa vaatimuksiin liittyvästä tiedosta oli tiimin sisäistämää asiakkaan kanssa keskustelun kautta syntynyttä hiljaista tietoa. Tätä kehitystiimin ja asiakkaan sisäistämää käsitystä vaatimuksista olisi voitu esimerkiksi haastatteluiden avulla kartoittaa tarkemmin ja selvittää, kuinka eroavat käsitykset vaatimuksista ja niiden prioriteeteista projektiin osallistuvilla lopulta oli.

8 Johtopäätökset

Tämä tutkielma käsitteli vaatimusmäärittelyä ketterässä ohjelmistoprojektissa. Luku 3 esitteli vaatimusmäärittelyä perinteisen ohjelmistokehityksen näkökulmasta sekä määritteli, mitä ohjelmiston vaatimuksella tarkoitetaan ja millaisista vaiheista vaatimusmäärittely koostuu. Luvussa 4 kuvailtiin ketterää ohjelmistokehitystä ja vaatimusmäärittelyn toteutumista siinä. Lisäksi selvitettiin, millaisia hyötyjä ketterällä vaatimusmäärittelyllä voidaan saavuttaa perinteiseen vaatimusmäärittelyyn verrattuna sekä millaisia haasteita ketterässä vaatimusmäärittelyssä voidaan kohdata. Lopuksi käytiin läpi tutkimuskirjallisuudessa tunnistettuja ketterän vaatimusmäärittelyn käytänteitä sekä selvitettiin, millaisia vaikutuksia näillä käytänteillä on ohjelmistoprojektien vaatimuksiin liittyviin riskeihin. Luvussa 5 esiteltiin tapaustutkimuksen aineisto ja käytetyt tutkimusmenetelmät. Luvussa 6 syvennyttiin tapaustutkimuksen tuloksiin. Luvussa 7 esitettiin pohdinta tulosten merkityksestä.

Tässä tutkielman päättävässä luvussa esitetään tutkielman johtopäätökset. Alaluvussa 8.1 käydään läpi vastaukset tutkielman alussa asetettuihin tutkimuskysymyksiin. Alaluku 8.2 listaa tähän tutkielmaan liittyvät rajoitteet. Lopuksi alaluvussa 8.3 pohditaan vielä ideoita mahdolliselle jatkotutkimukselle.

8.1 Vastaukset tutkimuskysymyksiin

Tämän tutkielman tutkimusongelma oli: *Miten vaatimukset muodostuvat ketterässä ohjelmistokehityksessä?* Tutkittava pääongelma jaettiin tarkempiin tutkimuskysymyksiin, joihin voidaan vastata tutkielmassa esitetyn aiemman tutkimuskirjallisuuden ja tapaustutkimusten tulosten avulla seuraavasti:

Tutkimuskysymys 1: Mitä ketterällä ohjelmistokehityksellä tarkoitetaan?

Ohjelmistokehitysmenetelmä voidaan määritellä ketteräksi sen ollessa inkrementaalinen, yhteistyöhön kannustava, suoraviivainen ja mukautuva. Ketteriä menetelmiä käyttäen ohjelmistoa tuotetaan useissa lyhyissä iteraatioissa, joiden päätteeksi toimitetaan aina uusi versio ohjelmistosta. Asiakas ja kehitystiimi työskentelevät yhdessä jatkuvasti tiiviisti kommunikoiden. Suora ja toimiva kommunikaatio on yksi tärkeimmistä tekijöistä ketterien oh-

jelmistoprojektien onnistumiselle. Tieto liikkuu kommunikaation avulla ja sitä käyttäen selvitetään ja määritellään projektin tavoitteet sekä ratkaistaan projektin aikana ilmenevät ongelmat. Ketterät menetelmät ovat kevyitä, yksinkertaisia ja nopeita. Prosessit ovat helposti opittavissa ja niitä on helppo soveltaa ja muokata vastaamaan yksittäisen projektin tarpeita.

Ketterät menetelmät myös pyrkivät koko projektin ajan reagoimaan hallitusti ja nopeasti muutostarpeisiin, jotka kohdistuvat kehitettävään ohjelmistoon, sen vaatimuksiin sekä prosessiin itseensä. Varautumalla muutokseen pyritään tuottamaan asiakkaan tarpeet täyttävä ohjelmisto. Ketterät menetelmät soveltuvat erityisesti projekteihin, joissa ei etukäteen pystytä tarkasti määrittelemään kaikkia ohjelmiston vaatimuksia tai projekteihin, joissa voidaan olettaa tiedossa olevien vaatimusten muuttuvan projektin edetessä.

Yksi tärkeistä päämääristä ketterissä menetelmissä on tuottaa arvoa jo heti projektin varhaisista vaiheista alkaen. Tämä saavutetaan toimittamalla tiheästi uusia versioita ohjelmistosta, jotka sisältävät asiakkaan kyseiseen versioon priorisoimat ominaisuudet. Ketterät menetelmät ovat parhaimmillaan projekteissa, joissa rakennettaan pieniä ohjelmistoa pienissä tiimeissä.

Tutkimuskysymys 2: Miten vaatimusmäärittely toteutuu ketterässä ohjelmistokehityksessä?

Ketterässä ohjelmistoprojektissa toteutuvaa vaatimusmäärittelyä kutsutaan myös ketteräksi vaatimusmäärittelyksi. Sen avulla on mahdollista suorittaa kaikkia perinteisen vaatimusmäärittelyn vaiheita eli vaatimusten esillesaanti, analyysi ja neuvottelu, dokumentointi, validointi sekä hallinta. Vaiheita ei suoriteta peräkkäin vaan iteratiivisesti jokaisen usean lyhyen kehityssyklin aikana. Vaiheet myös sekoittuvat keskenään ketterässä vaatimusmäärittelyssä, koska käytänteet poikkeavat perinteisistä ja koska myös ketterät menetelmät ovat keskenään erilaisia.

Asiakkaan ja kehitystiimin välinen jatkuva ja läheinen kommunikaatio on ketterän vaatimusmääritellyn kannalta oleellisessa asemassa, sillä sitä hyödynnetään jokaisessa vaatimusmäärittelyprosessin vaiheessa. Ketterässä ohjelmistokehityksessä vaatimusten esillesaanti tapahtuu iteratiivisesti läpi koko projektin. Vaatimuksia löydetään asiakkaan ja kehitystiimin välisen keskustelun avulla. Vaatimusten analysointi ja neuvottelu näkyvät siten, että löydettyjä vaatimuksia parannellaan, muutetaan ja priorisoidaan jokaisen kehityssyklin aikana. Tieto välittyy dokumentaation sijaan keskustelemalla eri osapuolten välillä. Vaatimukset kirjataan vapaamuotoisemmin esimerkiksi käyttäjätarinoina jatkuvasti priorisoituun kehitysjonoon. Vaatimusten validointi tapahtuu asiakkaan toimesta katsel-

mointipalaverien aikana sekä myös suunniteltaessa uutta iteraatiota ja sen aikana.

Ketterää vaatimusmäärittelyä tehdään vaatimusmäärittelyprosessia tukevien käytänteiden avulla. Käytänteitä on tunnistettu tutkimuskirjallisuudessa useita. Tässä tutkielmassa niistä esiteltiin kasvokkain käytävä kommunikaatio, iteratiivinen vaatimusmäärittely, jatkuva vaatimusten priorisointi, jatkuva suunnittelu, prototyypitys sekä katselmointipalaverit ja hyväksymistestaus.

Ketterän vaatimusmäärittelyn käytänteillä on myös havaittu olevan vaikutus ohjelmistoprojektien vaatimuksiin liittyviin riskeihin. Vaatimusten puuttumiseen ja tärkeiden vaatimusten alhaiseen prioriteettiin liittyvää riskiä voidaan pienentää ketterän vaatimusmäärittelyn käytänteillä. Pyrkimys tehdä vaatimuksista kattavia ennen toteutusta on riski, jota ketterällä vaatimusmäärittelyllä voidaan myös pienentää. Ketterällä vaatimusmäärittelyllä on puolestaan suurentava vaikutus seuraaviin riskeihin: toiminnallisten vaatimusten ylikorostuminen, riittämätön vaatimusten tarkastus sekä vaatimusten esittäminen toteutuksen muodossa. Ottamalla vaatimuksiin liittyvät riskit huomioon voidaan tehdä perustellumpia valintoja projektin vaatimusmäärittelyn käytänteistä sekä sopivasta prosessimallista.

Tutkimuskysymys 3: Mitä hyötyjä ja haasteita ketterällä vaatimusmäärittelyllä voidaan havaita perinteisiin menetelmiin verrattuna?

Ketterällä vaatimusmäärittelyllä on esitetty olevan mahdollista saavuttaa kuusi erilaista hyötyä perinteiseen vaatimusmäärittelyyn verrattuna. Nämä hyödyt ovat vähemmän prosessin vuoksi tehtävää työtä, parantunut ymmärrys vaatimuksista, vähentynyt ylikuormittaminen, muutostarpeeseen vastaaminen, nopeat toimitukset ja validaatio sekä parantuneet asiakassuhteet. Tapaustutkimuksen avulla voitiin empiirisesti havaita edellä mainittujen ketterän vaatimusmäärittelyn hyötyjen toteutuminen ketterällä menetelmällä startup-yritykselle toteutetussa ohjelmistoprojektissa.

Ketterällä vaatimusmäärittelyllä voidaan kohdata ohjelmistoprojekteissa myös haasteita, joilla voi olla negatiivinen vaikutus projektin tulokseen. Tutkimuskirjallisuudessa on esitetty seuraavat haasteet: asiakkaaseen liittyvät ongelmat, käyttäjätarinoiden riittämättömyys, haasteet vaatimusten priorisoinnissa, kasvava tekninen velka, tukeutuminen hiljaiseen tietoon vaatimuksista sekä epätarkat työmääräarviot. Suoritetun tapaustutkimuksen aineistoa analysoimalla pystyttiin projektissa havaitsemaan epätarkkoihin työmääräarvioihin liittyvä haaste.

Tästä voidaan päätellä, että hyötyjen saavuttaminen on mahdollista pienellä tiimillä toteutettavassa asiakasprojektissa, jossa ei ole etukäteen tarkasti määriteltyjä ohjelmiston

vaatimuksia. Toisaalta haasteitakin voidaan kohdata jo hyvin lyhyessä ajassa, vaikka rakennettaisiin pientä ja yksinkertaista sovellusta.

8.2 Rajoitteet

Suoritettuun tutkimukseen saattaa liittyä muutamia rajoitteita, jotka on syytä ottaa huomioon tuloksia tulkittaessa, sillä ne voivat vaikuttaa tulosten validiteettiin.

Tutkielman kirjoittaja osallistui opiskelijana tapaustutkimuksessa tarkasteltavalle kurssille, jolta tapaustutkimuksen materiaali on kerätty. Hän keräsi itse muistiinpanoja kurssilla tehdyn projektin aikana. On olemassa riski, että projektin jälkeen aineiston pohjalta tehdyt tulkinnat saattavat olla osittain puolueellisia. Osallistuvan havainnoinnin tukena oli kuitenkin myös muina tietolähteinä Kanban-taulun läpikäyneet tehtävät (post-it-laput), kurssin päätteeksi suoritetun debriefing-session aikana yhdessä asiakkaan kanssa tuotettu kuva ja Git-versionhallinnan sisältämä koodi muutoksineen. Täten analyysin aikana on voitu vertailla samaa tietoa useista eri tietolähteistä.

Tämän tapaustutkimuksen kannalta tässä projektissa kerättiin muistiinpanoja ja muuta aineistoa laajalla otannalla eli ei pelkästään pitäen silmällä juuri tämän tutkielman tulosten analysoinnin tarpeita. Osa analysoiduista tuloksista on siis mahdollisesti tehty osittain muistinvaraisesti. Tapaustutkimuksen aineistonkeruuvaiheessa tutkielmassa käytettävä aineiston analysointitapa ei ollut kirjoittajalle vielä täysin selvä.

Tapaustutkimuksen projektiin osallistuneet henkilöt olivat kaikki opiskelijoita. Kaikki olivat toisaalta jo opintojensa loppuvaiheessa ja monella oli myös työkokemusta. Tutkimus suoritettiin myös ympäristössä, joka jäljittelee oikeaa työympäristöä, mutta ei välttämättä pysty kaikilta osin mallintamaan siihen liittyviä muuttujia. Tiimi pystyi kuitenkin työskentelemään itsenäisesti asiakkaan kanssa ilman muita projektia ohjaavia tahoja. Projektista tehtiin ainoastaan suoria havaintoja kirjoittajan suorittaman osallistuvan havainnoinnin lisäksi, joten näillä ei pitäisi olla vaikutusta lopputulokseen.

8.3 Mahdolliset jatkotutkimuskohteet

Tutkielman hyödyntämään tutkimuskirjallisuuteen ja tapaustutkimukseen liittyen voidaan esittää muutama potentiaalinen jatkotutkimusaihe. Ensinnäkin uusia tutkimuskirjallisuudessa mainittuja ketterän vaatimusmäärittelyn käytänteitä olisi hyödyllistä kartoittaa ja

tutkia niiden vaikutuksia ohjelmistoprojektin vaatimuksiin liittyviin riskeihin. Tällä tavalla saataisiin käyttöön suurempi määrä erilaisia käytänteitä, joiden hyödyntämistä projekteissa voitaisiin arvioida samantapaisia kriteereitä käyttäen.

Tämän tutkielman tapaustutkimus suoritettiin ohjelmistoprojektissa, jossa kehitystiimi koostui kokonaan opiskelijoista ja suurimmalla osalla ei ollut vuosien mittaista työkokemusta pidempiaikaisissa ohjelmistoprojekteissa työskentelemisestä. Tapaustutkimuksen suorittaminen todellisissa työelämän projekteissa voisi olla perusteltu, jotta saataisiin tuloksia myös ammattilaisilla suoritetusta pidempikestoisista projekteista sekä erilaisista työympäristöistä.

Tutkielman tapaustutkimuksessa pystyttiin havaitsemaan ainoastaan yksi ketterään vaatimusmäärittelyyn liittyvistä haasteista (epätarkat työmääräarviot). Pienissä ja lyhyissä projekteissa haasteet eivät välttämättä ehdi nousta esille, joten voisi olla hyödyllistä selvittää miten haasteet ilmenevät suurissa ja pitkäkestoisissa ohjelmistoprojekteissa sekä minkälaisilla tavoilla haasteita on pyritty näissä projekteissa ratkaisemaan.

Lähteet

Abrahamsson, P., Salo, O., Ronkainen, J. ja Warsta, J. (2002). *Agile Software Development Methods: Review and analysis*. VTT Publications.

Abrahamsson, P., Warsta, J., Siponen, M. T. ja Ronkainen, J. (2003). "New directions on agile methods: a comparative analysis". Teoksessa: *Software Engineering, 2003. Proceedings. 25th International Conference on*, s. 244–254. DOI: 10.1109/ICSE.2003.1201204.

Agile Alliance (2001). *Manifesto for Agile Software Development*. URL: https://agilemanifesto.org/ (viitattu 14. 04. 2020).

Beck, K. (2000). *Extreme programming explained: embrace change*. eng. Reading (MA): Addison-Wesley, 190 s.

Bjarnason, E., Wnuk, K. ja Regnell, (2011). "A Case Study on Benefits and Side-Effects of Agile Practices in Large-Scale Requirements Engineering". Teoksessa: *Proceedings of the 1st Workshop on Agile Requirements Engineering*. AREW '11. Lancaster, United Kingdom: Association for Computing Machinery. DOI: 10.1145/2068783.2068786.

Boehm, B. (2000). "Requirements that handle IKIWISI, COTS, and rapid change". *Computer* 33.7. ID: 1, s. 99–102.

Boehm, B. W. (1988). "A spiral model of software development and enhancement". *Computer* 21.5, s. 61–72.

Boehm, B. ja Turner, R. (2003). *Balancing agility and discipline: A guide for the perplexed*. Addison-Wesley Professional.

Cao, L. ja Ramesh, B. (2008). "Agile Requirements Engineering Practices: An Empirical Study". *Software, IEEE* 25.1, s. 60–67. DOI: 10.1109/MS.2008.1.

Dybå, T. ja Dingsøyr, T. (elokuu 2008). "Empirical studies of agile software development: A systematic review". *Information and Software Technology* 50.9, s. 833–859. DOI: 10.1016/j.infsof.2008.01.006.

Hannay, J. E., Dybå, T., Arisholm, E. ja Sjøberg, D. I. K. (heinäkuu 2009). "The effectiveness of pair programming: A meta-analysis". *Information and Software Technology* 51.7, s. 1110–1122. DOI: 10.1016/j.infsof.2009.02.001.

Hansen, S., Berente, N. ja Lyytinen, K. (2009). "Requirements in the 21st century: Current practice and emerging trends". Teoksessa: Design Requirements Engineering: A Ten-Year Perspective. Springer, s. 44–87.

Heikkilä, V. T., Damian, D., Lassenius, C. ja Paasivaara, M. (2015). ”A Mapping Study on Requirements Engineering in Agile Software Development”. Teoksessa: s. 199–207. DOI: 10.1109/SEAA.2015.70.

Hirsjärvi, S., Remes, P. ja Sajavaara, P. (2000). *Tutki ja kirjoita*. 6.-9., uud. p. Helsinki: Tammi.

Inayat, I., Salim, S. S., Marczak, S., Daneva, M. ja Shamshirband, S. (lokakuu 2015). ”A systematic literature review on agile requirements engineering practices and challenges”. *Computers in Human Behavior* 51, s. 915–929. DOI: 10.1016/j.chb.2014.10.046.

ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary (2017). ID: 1. DOI: 10.1109/IEEESTD.2017.8016712.

Kotonya, G. ja Sommerville, I. (1998). *Requirements Engineering: Processes and Techniques*. 1st. Wiley Publishing.

Lamsweerde, A. van (2009). *Requirements engineering: from system goals to UML models to software specifications*. eng. Reprinted 2010. Chichester: Wiley, 682 s.

Lindvall, M., Basili, V., Boehm, B., Costa, P., Dangle, K., Shull, F., Tesoriero, R., Williams, L. ja Zelkowitz, M. (2002). ”Empirical findings in agile methods”. Teoksessa: Extreme Programming and Agile Methods—XP/Agile Universe 2002. Springer, s. 197–207.

Lucassen, G., Dalpiaz, F., Werf, J. M. E. M. van der ja Brinkkemper, S. (2015). ”Forging high-quality User Stories: Towards a discipline for Agile Requirements”. Teoksessa: *2015 IEEE 23rd International Requirements Engineering Conference (RE)*. ID: 1, s. 126–135. DOI: 10.1109/RE.2015.7320415.

Lucia, A. ja Qusef, A. (2010). ”Requirements Engineering in Agile Software Development”. *Journal of Emerging Technologies in Web Intelligence* 2.3. DOI: 10.4304/jetwi.2.3.212-220.

Orr, K. (2004). ”Agile requirements: opportunity or oxymoron?” *Software, IEEE* 21.3. ID: 1, s. 71–73.

Paetsch, F., Eberlein, A. ja Maurer, F. (2003). ”Requirements engineering and agile software development”. Teoksessa: *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on*, s. 308–313.

Ramesh, B., Cao, L. ja Baskerville, R. (2010). ”Agile requirements engineering practices and challenges: an empirical study”. *Information Systems Journal* 20.5, s. 449–480. DOI: 10.1111/j.1365-2575.2007.00259.x.

Royce, W. W. (1970). ”Managing the development of large software systems”. Teoksessa: *proceedings of IEEE WESCON*. Vol. 26. Los Angeles. Luku 8.

Runeson, P. ja Höst, M. (huhtikuu 2009). ”Guidelines for conducting and reporting case study research in software engineering”. English. *Empirical Software Engineering* 14.2. J2: Empir Software Eng, s. 131–164. DOI: 10.1007/s10664-008-9102-8.

Savolainen, J., Kuusela, J. ja Vilavaara, A. (2010). ”Transition to Agile Development - Rediscovery of Important Requirements Engineering Practices”. Teoksessa: *2010 18th IEEE International Requirements Engineering Conference*, s. 289–294. DOI: 10.1109/RE.2010.41.

Schwaber, K. ja Sutherland, J. (2017). ”The Scrum Guide”. URL: https://www.scrumguides.org/ (viitattu 14. 10. 2020).

Sillitti, A., Ceschi, M., Russo, B. ja Succi, G. (2005). ”Managing uncertainty in requirements: a survey in documentation-driven and agile companies”. Teoksessa: *Software Metrics, 2005. 11th IEEE International Symposium*, s. 10–17.

Software Factory – University of Helsinki (2018). URL: https://www.softwarefactory.cc/ (viitattu 14. 04. 2020).

Sommerville, I. (2007). *Software Engineering*. 8th. Pearson Addison Wesley.

Sommerville, I. ja Sawyer, P. (1997). *Requirements engineering: a good practice guide*. John Wiley Sons, Inc.

Sutherland, J. ja Schwaber, K. (2012). ”The Scrum Papers: Nuts, Bolts, and Origins of an Agile Method”. URL: http://jeffsutherland.com/scrum/ScrumPapers.pdf (viitattu 14. 10. 2020).

Turk, D., France, R. ja Rumpe, B. (lokakuu 2005). ”Assumptions Underlying Agile Software-Development Processes”. *Journal of Database Management* 16.4, s. 62–87. DOI: 10.4018/jdm.2005100104.

Webster, J. ja Watson, R. T. (huhtikuu 2002). ”Analyzing the Past to Prepare for the Future: Writing a Literature Review”. *MIS Quarterly* 26.2. 16, s. xiii–xxiii. URL: http://www.jstor.org/stable/4132319.